APPENDIX I: Prior art submitted by submitter

The following table comprises a number of papers that the submitter believes should have been disclosed in the IDS for the application and that are directly relevant to the patentability of the presently pending claims.

Our Reference	Date yyyy-mm-dd	Publication	Title	Authors	Location
OCCA	2001-05-21	Proceedings of the 2001 ACM SIGMOD international conference on Management of data	On computing correlated aggregates over continual data streams	Johannes Gehrke, Cornell University Flip Korn, AT&T Labs-Research Divesh Srivastava, AT&T Labs-Research	Santa Barbara, California Pages: 13 - 24 ISSN: 0163-5808
FTSA	2001-06-XX	Proceedings of the 18th International Conference on Data Engineering	Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data	Samuel Madden (madden@cs.berkeley.edu) Michael J. Franklin (franklin@cs.berkeley.edu)	Page: 555 ISSN:1063-6382
MSSO	2001-07-XX	Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms	Maintaining stream statistics over sliding windows	Mayur Datar, Stanford University, Stanford CA Aristides Gionis, Stanford University, Stanford CA Piotr Indyk, MIT Laboratory for Computer Science, Cambridge, Mass. Rajeev Motwani, Stanford University, Stanford CA	San Francisco, California Pages: 635 - 644 ISBN:0-89871-513-X

CQDS	2001-09-XX	SIGMOD Record, Sept. 2001	Continuous Queries over Data Streams	Babu, Shivnath; Widom, Jennifer	Volume 30 , Issue 3 (September 2001) Pages: 109 - 120 ISSN:0163-5808
MIDS	2002-03-09	Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems	Models and issues in data stream systems	Brian Babcock Shivnath Babu Mayur Datar, Rajeev Motwani, Jennifer Widom Stanford University, Stanford, CA	Pages: 1 - 16 ISBN:1-58113-507-6
CS02	2002-02-XX	Technical Report CS-02-01, Department of Computer Science, Brown University, Feb. 2002.		Don Carney, Ugur Cetintemel, Mitch Cherniack, Slangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, Stan Zdonik	
CACQ	2002-06-04	Proceedings of the 2002 ACM SIGMOD international conference on Management of data	Continuously adaptive continuous queries over streams	Samuel Madden, UC Berkeley Mehul Shah, UC Berkeley Joseph M. Hellerstein, UC Berkeley Vijayshankar Raman, IBM Almaden Research Center	Pages: 49 - 60 ISBN:1-58113-497-5
EWJU	2002-08-20	Proceedings of the 28th VLDB Conference 2002, Hong Kong,	Evaluating Window Joins Over Unbounded	Jaewoo Kang Jeffrey F. Naughton	

		China	Streams	Stratis D. Viglas	
MSNC	2002-08-20 This paper based on and is substantially similar to "CS02" above which is dated 2002-02-XX	Proceedings of the 28th VLDB Conference 2002, Hong Kong, China	Monitoring Streams – A New Class of Data Management Applications	Don Carney Ugur Cetintemel Mitch Cherniack Christian Convey Sangdon Lee Greg Seidman Michael Stonebraker Nesime Tatbul Stan Zdonik	
SQSD	2002-08-20	Proceedings of the 28th VLDB Conference 2002, Hong Kong, China	Streaming Queries over Streaming Data	Sirish Chandrasekaran Michael J. Franklin	

APPENDIX II: Claims as pending in 10/775,744

1. A method for monitoring time series, comprising: a means for storing data from multiple input time series; a means for executing persistent queries; a means for dynamic management of persistent queries; a means for performing online computation of statistics.

2. The method as recited in claim 1, further comprising: a means for dynamic management of windows.

3. The method as recited in claim 1, further comprising: a means for using historical values in present windows to help populate inserted windows.

Fjording the Stream: An Architecture for Queries over Streaming Sensor Data

Samuel Madden
madden@cs.berkeley.edu

Michael J. Franklin
franklin@cs.berkeley.edu

Abstract

If industry visionaries are correct, our lives will soon be full of sensors, connected together in loose conglomerations via wireless networks, each monitoring and collecting data about the world at large. These sensors behave very differently from traditional database sources: they have intermittent connectivity, are limited by severe power constraints, and typically sample periodically and push immediately, keeping no record of historical information. These limitations make traditional database systems inappropriate for queries over sensors. We present the Fjords architecture for managing multiple queries over many sensors, and show how it can be used to limit sensor resource demands while maintaining high query throughput. We evaluate our architecture using traces from a network of traffic sensors deployed on Interstate 80 near Berkeley and present performance results that show how query throughput, communication costs, and power consumption are necessarily coupled in sensor environments.

1 Introduction

Over the past few years, a great deal of attention in the networking and mobile-computing communities has been directed towards building networks of ad-hoc collections of sensors scattered throughout the world. MIT, UCLA, and UC Berkeley [21, 24, 11, 31] have all embarked on projects to produce small, wireless, battery-powered sensors and low-level networking protocols. These projects have brought us close to the the vision of ubiquitous computing [38], in which computers and sensors assist in every aspect of our lives. To fully realize this vision, however, it will be necessary to combine and query the sensor readings produced by these collections of sensors. Sensor networks will produce very large amounts of data, which needs to be combined and aggregated to analyze and react to the world. Clearly, the ability to apply traditional data-processing languages and operators to this sensor data is highly desirable. Unfortunately, standard DBMS assumptions about the characteristics of data sources do not apply to sensors, so a significantly different architecture is needed.

There are two primary differences between sensor based data sources and standard database sources. First, sensors typically deliver data in *streams*: they produce data continuously, often at well defined time intervals, without having been explicitly asked for that data. Queries over those streams need to be processed in near real-time, partly because it is often extremely expensive to save raw sensor streams to disk, and partly because sensor streams represent real-world events, like traffic accidents and attempted network break-ins, which need to be responded to. The second major challenge with processing sensor data is that sensors are fundamentally different from the over-engineered data-sources typical in a business DBMS. They do not deliver data at reliable

rates, the data is often garbled, and they have limited processor and battery resources which the query engine needs to conserve whenever possible.

Our contribution to the problem of querying sensor data operates on two levels: First, we propose an enhanced query plan data-structure called Fjords ("Framework in Java for Operators on Remote Data Streams"), which allows users to pose queries that combine streaming, push-based sensor-sources with traditional sources that produce data via a blocking, pull-based iterator interface. To execute these queries, our system provides non-blocking and windowed operators which are suited to streaming data. Second, we propose power-sensitive Fjord operators called *sensor-proxies* which serve as mediators between the query processing environment and the physical sensors.

A key component of Fjords is that data flows into them from sensors and is pushed into query operators. Operators do not actively pull data to process, rather, they operate on the samples when sensors make them available and are otherwise idle. They never wait for a particular tuple to arrive. Because of this passive-behavior, the adaptive-query processing situation of an operator being "blocked", waiting for input, does not arise.

Sensor data processing and the related area of query processing over data streams have been the subject of increasing attention recently. Research projects on sensor networks have begun to recognize the importance of data processing issues such as aggregating readings from multiple sensors [19, 22]. In the database community, there has been significant work on query operators and evaluation algorithms for processing queries over streams of data [35, 33, 30, 12]. In addition, the Cougar project at Cornell has been exploring the use of Object-Relational abstractions for querying sensor-based systems [26]. These projects and other related efforts are providing key technology that will be necessary for data intensive sensor-based applications. Our work differs, however, in that it is focused on providing the *underlying systems architecture* for sensor data management. Thus, our focus is on the efficient, adaptive, and power sensitive infrastructure upon which these new query processing approaches can be built. To our knowledge, this is the first work addressing the low level database engine support required for sensor-centric data-intensive systems. We address related work in more detail in Section 6.

We now present an overview of the sensor-query processing environment and discuss the sensor testbed which we are building. In the remaining sections, we present the specific requirements of sensor query processing, propose our solutions for satisfying those requirements, present some initial performance results, discuss related work, offer directions for future work and conclude.

2 Background

2.1 Sensor Environment

In this paper, we focus on a specific type of sensor processing environment in which there are a large number of fairly simple sensors over which users want to pose queries. For our purposes, a sensor consists of a remote measurement device that provides data at regular intervals. A sensor may have some limited processing ability or configurability, or may simply output a raw stream of measurements. Because sensors have at best limited capabilities, we do not ask them to parse queries or keep track of which clients need to receive samples from them: they simply sample data, aggregate that data into larger packets, and route those packets to a data processing node, which is a fixed, powered, and well-connected server or workstation with abundant disk and memory

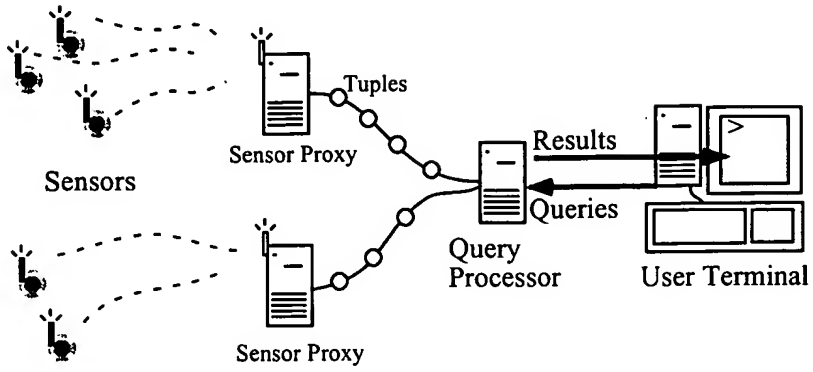


Figure 1: *Environment For Sensor Query Processing*

resources, such as would be used in any conventional database system. We call the node that receives sensor data the sensor's *proxy*, since it serves as that sensor's interface into the rest of the query processor. Typically, one machine is the proxy for many sensors. The proxy is responsible for packaging samples as tuples and routing those tuples to user queries as needed. The general query environment is shown in Figure 1. Users issue queries to the server; the server processes the query, instantiates operators and locates sensor proxies, and starts the flow of tuples. Although sensors do not directly participate in query processing, their proxy can adjust their sample rate or ask them to perform simple aggregation before relaying data, which, as we will show, is an important aspect of efficiently running queries over many sensors.

We are building this system as a part of the Telegraph [17] data flow processing engine which is under development at UC Berkeley. We have extended this system with our Fjords data-flow architecture. In Telegraph, users pose queries at a workstation on which they expect results to appear. That workstation translates queries into Fjords through a process analogous to normal query optimization. New Fjords may be merged into already running Fjords with similar structures over the same sensors, or may run independently. Queries run continuously because streams never terminate; queries are removed from the system only when the user explicitly ends the query. Results are pushed from the sensors out toward the user, and are delivered as soon as they become available.

Information about available sensors in the world is stored in a catalog, which is similar to a traditional database catalog. The data that sensors provide is assumed to be divisible into a typed schema, which users can query much as they would any other relational data source. Sensors submit samples, which are keyed by sample time and logically separated into fields; the proxy converts those fields into native database tuples which local database operators understand. In this way, sensors appear to be standard object-relational tables; this is a technique proposed in the Cougar project at Cornell[26], and is consistent with Telegraph's view of other non-traditional data sources, such as web pages, as relational tables.

2.1.1 Traffic Sensor Testbed

We have recently begun working with the Berkeley Highway Lab (BHL), which, in conjunction with the California Department of Transportation (CalTrans), is deploying a sensor infrastructure on Bay Area freeways to

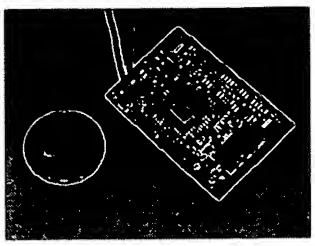


Figure 2: *A TinyOS Sensor Mote*

monitor traffic conditions. The query processing system we present is being built to support this infrastructure. Thus, in this paper, we will use traffic scenarios to motivate many of our examples and design decisions.

CalTrans has embedded thousands primitive sensors on Bay Area highways over the past few decades. These sensors consist of inductive loops that register whenever a vehicle passes over them, and can be used to determine aggregate flow and volume information on a stretch of road as well as provide gross estimates of vehicle speed and length. Typically these loops are used to monitor specific portions of the highway by placing data-collection hardware at sites of interest. This data is not available for real-time processing; data is simply collected and stored for later, offline analysis of flow patterns.

In the near future, it is expected that real-time sensor data will be widely available, as CalTrans is working to deploy intelligent sensors which can relay information from all over the Bay Area to its own engineers and the BHL. With several thousand sensors streaming data, efficient techniques for executing queries over those streams will become crucial. The exact nature of this new generation of sensors has not yet been determined, but a number of groups around the country are working on architectures and operating systems for sensors which are very well-suited to this type of environment [21, 24, 11, 31].

2.1.2 Next-Generation Traffic Sensors

Current research on sensor devices is focused on producing very small sensors that can be deployed in harsh environments (e.g. the surface of a freeway.) Current sensor prototypes at UC Berkeley, MIT, and UCLA share similar properties: they are very small, wireless radio-driven, and battery powered, with a current size of about 10^3cm. The ultimate goal is to produce devices in the 1^3mm range – about the size of a gnat [21]. We refer to such miniscule sensors as "motes". Current prototypes use 8bit microprocessors with small amounts of RAM (less than 32kBytes) running from 1 to 16 MHz, with a small radio capable of transmitting at tens of kilobits per second with a range of a few hundred feet[18]. Figure 2 shows an example of such a sensor from the TinyOS project at Berkeley.

As sensors become miniaturized, they will be small enough that they could be scattered over the freeway such that they will not interfere with cars running over them, or could fit easily between the grooves in the roadway. Sensors could be connected to existing inductive loops, or be augmented with light or pressure sensors that could detect whenever a car passed over them, inferring location information from the latencies of the sensors relative to the fixed base stations[28]. Motes would use radios to relay samples to nearby wireless basestations, which would, in turn, forward them to query processing workstations.

One key component of this new generation of sensors is that they are capable of computation and remote reprogramming. This provides control over the source, quality, and density of samples. For example, two sensors in the same lane within a few feet of each other are virtually guaranteed to see the same set of cars, so asking both for the same information is unnecessary. However, there may be times when the two sensors can complement each other – for instance, the light sensor on one mote could be used to corroborate the reading of the pressure sensor from the other, or to verify that the other is functioning properly.

Similarly, the ability of motes to perform some computation and aggregation allows the computational burden on the server and the communications burden on the motes to be reduced. For example, rather than directly transmitting the voltage reading from a light sensor many times a second, motes could transmit a count of the number of cars which have passed over them during some time interval, reducing the amount of communication which is required and saving the central server from having to count the number of cars from the voltage readings. In an environment with tens of thousands of sensors, the benefits of such reductions can be substantial.

2.2 Requirements for Query Processing over Sensors

Given our desire to run queries over these traffic sensors, we now examine some of the issues posed by sensors that are not present in traditional data sources. In this section, we focus on the properties of sensors and streaming data which must be taken into account when designing the low level infrastructure needed to efficiently integrate streaming sensor data into a query processor. There are other important issues in sensor-stream processing, such as query language and algebra design and the detailed semantics of stream operators, which are not the focus of this work. We will discuss these in more detail in Section 6.

2.2.1 Limitations of Sensors

Limited resource availability is an inherent property of sensors. Scarce resources include battery capacity, communications bandwidth, and CPU cycles. Power is the defining limit: it is always possible to use a faster processor or a more powerful radio, but these require more power which often is not available. Current small battery technology provides about 100mAh of capacity. This is enough to drive a small Atmel processor, like the one used in several wireless sensor prototypes, at full speed for only 3.5 hours. Similarly, the current TRM-1000 radio, also used in many sensor prototypes, uses about 4 μJ per bit of data transmitted: enough to send just 14MB of data using such a battery. In practice, power required for sending data over the wireless radio is the dominant cost [27], so it is often worth spending many CPU cycles to conserve just a few bytes of radio traffic.

One principal way that battery power is conserved is by powering down, or *sleeping*, parts of sensors when they are not needed. It is very important that sensors maximize the use of these low power modes whenever possible. An easy way to allow sensors to spend more time sleeping is to decrease the rate at which sensors collect and relay samples. The database system must enable this by dynamically adjusting the sample rate based on current query requirements.

In addition to power limitations, sensors have the added feature that they include small processors which can be used to perform some processing on samples. Database systems need to able to take advantage of this processing ability, as it can dramatically reduce the power consumption of the sensors and reduce the load on the query processor itself.

2.2.2 Streaming Data

Another property of sensors is that they produce continuous, never ending streams of data (at least, until the sensors run out of battery power!). Any sensor query processing system needs to be able to operate directly on such data streams. Because streams are infinite, operators can never compute over an entire streaming relation: i.e. they cannot be *blocking*. Many implementations of traditional operators, such as sorts, aggregates, and some join algorithms therefore, cannot be used. Instead, the query processor must include special operators which deliver results incrementally, processing streaming tuples one at a time or in small blocks.

Streaming data also implies that sensors *push* data into a query plan. Thus, the conventional pull-based iterator [13] model does not map well onto sensor streams. Although possible, implementing a strict iterator model-like interface on sensors requires them to waste power and resources. To do so, sensors must keep the receivers on their radios powered up at all times, listening for requests for data samples from multiple user queries. Power, local storage, and communications limitations make it much more natural for sensors to deliver samples when those samples become available.

Since many sensors have wireless connections, data streams may be delivered intermittently with significant variability in available bandwidth. Even when connectivity is generally good, wireless sensor connections can be interrupted by local sources of interference, such as microwaves. Any sensor database-system needs to expect variable latencies and dropped or garbled tuples, which traditional databases do not handle. Furthermore, because of these high latencies, an operator looking for a sensor tuple may be forced to block for some time if it attempts to pull a tuple from the sensor. Thus, operators must process data only when sensors make it available.

2.2.3 Processing Complex Queries

Sensors pose additional difficulties in a query processing system which is running complex or concurrent queries. In many sensor scenarios, multiple users pose similar queries over the same data streams. In the traffic scenario, commuters will want to know about road conditions on the same sections of road, and so will issue queries against the same sensors. Since streams are append-only, there is no reason that a particular sensor reading should not be shared across many queries. As our experiments in Section 4.3 show, this sharing greatly improves the ability of a sensor-query system to handle many simultaneous queries.

Furthermore, the demands placed on individual sensors vary based on time of day, current traffic conditions, and user requirements. At any particular time users are very interested in some sensors, and not at all interested in others. A query processing system should be able to account for this by dynamically turning down the sample and data delivery rates for infrequently queried sensors.

Finally, users querying sensors need a way to name the sensors that exist in the world. Traditional databases assume that users know the names of the sources they wish to run queries over. In a sensor environment with thousands of dynamic sensor-streams, this is not a realistic assumption. Instead, users must be able query the catalog, looking for sources which have desired properties. For instance in our traffic testbed, users will need to find sensors which are near a specific the part of the freeway.

3 Solution

Having described the difficulties involved integrating sensor streams into a query processor, we now present our solution, which, as previously mentioned, consists of two core components: Fjords, an adaptive modular data-flow for combining streaming sensor data and traditional data sources, and the sensor-proxy, for mediating between sensors and the query-processor, while taking into account the specific limitations of sensors.

3.1 Fjords: Generalized Query Plans for Sensor Streams

A Fjord is a generalization of traditional approaches to query plans: operators export an iterator-like interface and are connected together via local pipes or wide area queues. Fjords, however, also provide support for integrating streaming data that is pushed into the system with disk-based data which is pulled by traditional operators. As we will show, Fjords also allow combining multiple queries into a single plan and explicitly handle operators with multiple inputs and outputs.

Previous database architectures are not capable of combining streaming and static data. They are either strictly pull-based, as with the standard iterator model, or strictly push based, as in parallel processing environments. We believe that the hybrid approach adopted by Fjords, whereby streams can be combined with static sources in a way which varies from query-to-query is an essential part of any data processing system which claims to be able to compute over streams.

Figure 3 shows a Fjord running across two machines, with the left side detailing the modules running on a local machine. Each machine involved in the query runs a single *controller* in its own thread. This controller accepts messages to instantiate *operators*, which include the set of standard database modules – join, select, project, and so on. The controller also connects local operators via *queues* to other operators which may be running locally or remotely. Queues export the same interface whether they connect two local operators or two operators running on different machines, thus allowing operators to be ignorant of the nature of their connection to remote machines. Each query running on a machine is allocated its own thread, and that thread is multiplexed between the local operators via procedure calls (in a pull-based architecture) or via a special *scheduler* module that directs operators to consume available inputs or to produce outputs if they are not explicitly invoked by their parents in the plan. The Fjord shown in in Figure 3 was instantiated by a message arriving at the local controller; it applies two predicates to a stream of tuples generated by joining a sensor stream with a remote disk source.

In the rest of this section, we discuss the specific architecture of operators and queues in Fjords. We discuss specific operators for streams and present our power sensitive operator, the sensor proxy. We then describe how the connection of those operators via queues forms a Fjord, and show how Fjords can be used to share the work of multiple queries.

3.1.1 Operators and Queues

Operators form the core computational unit of Fjords. Each operator O has a set of input queues, Q_i and a set of output queues Q_o. O reads tuples in any order it chooses from Q_i and outputs any number of tuples to some or all of the queues in Q_o. This definition of operators is intentionally extremely general: Fjords are a dataflow architecture that is suitable for building more than just traditional query plans. For instance, in Section 4 we discuss folding multiple queries into a single Fjord; this is accomplished by creating operators with multiple

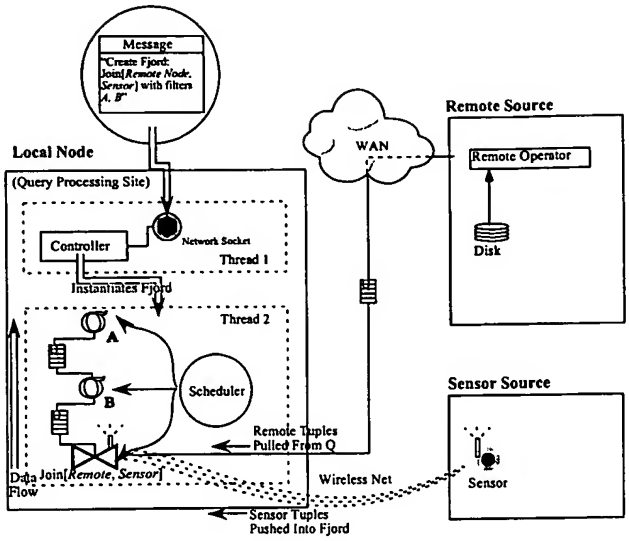


Figure 3: *A Fjord:* Join[*Remote Node,Sensor*] with filters *A* and *B*

outputs. These outputs route results of a single select or join, or an entire query, to multiple upstream operators or end users.

Queues are responsible for routing data from one operator (the *input operator*) to another (the *output operator.*) Queues have only one input and one output and perform no transformation on the data they carry. Like operators, queues are a general structure. Specific instances of queues can connect local operators or remote operators, behave in a push or pull fashion, or offer transactional properties which allow them to guarantee exactly-once delivery.

Naturally, it is possible for a queue to fill. When this happens, one has no choice but to discard some data: as is the case with network routers and multimedia streams, it is not possible to pause a stream of real-world data. The choice of which data to discard is application dependent, although with sensors typically the oldest samples should be thrown out first, as they are least relevant to the current state of the sensor.

3.1.2 Flexible Data Flow

The key advantage of Fjords is that they allow distributed query plans to use a mixture of push and pull connections between operators. Push or pull is implemented by the queue: a push queue relies on its input operator to *put* data into it which the output operator can later *get*. A pull queue actively requests that the input operator produce data (by calling its *transition* method) in response to a *get* call on the part of the output operator (see Figure 5b).

Push queues make it possible for sensor streams to work. When a sensor tuple arrives at a sensor-proxy, that proxy pushes those tuples onto the input queues of the queries which use it as a source. The operators draining those queues never actively call the sensor proxy; they merely operate on sensor data as it is pushed into them.

Figure 4 illustrates a join operator over two sensor streams which works by "zippering" together the streams

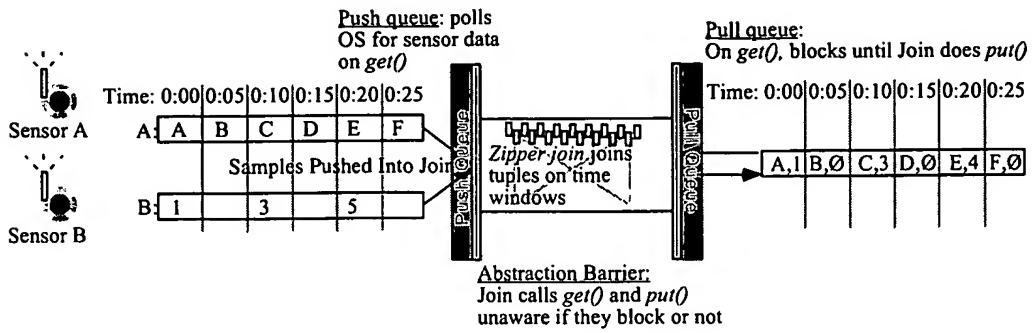


Figure 4: *Zipper Join Operator and Queues*

according to a time interval. [1] The join is not aware whether its inputs are being pushed in as streams or being pulled from disk. Similarly, it does not know whether its output is being pushed into or pulled by its parent operator: the queues on either side act as an abstraction barrier which allow it to function properly irrespective of the data flow properties of its children and parent. In the example here, the pull queues on the input poll (without blocking) the operating system to see if data is ready. The join alternates between checking these inputs for data. If data is pushed into one of these queues, the join checks to see if a matching tuple from the other stream has arrived in the current time window. If one has, the two tuples are joined and enqueued on the output queue. Otherwise, the join waits until the end of the current time window and performs an outer-join with the one result tuple and the null value. When the join's parent tries to dequeue a result, the output pull queue blocks and repeatedly schedules the zipper join until it produces a result.

We now present a detailed view of the internals of a Fjord operator.

3.1.3 State Based Execution Model

The programming model for operators is based upon state machines: each operator in the query plan represents a state in a transition diagram, and as such, an operator is required to implement only a single method: $o \leftarrow transition[s, i]$ which, given a current state s and some set of inputs i causes the operator to transition to a new state (or remain in the same state) and possibly produce some set of output tuples o. Implicit in this model of state programming is that operators do not block: when an operator needs to poll for data, it checks its input queue once per *transition* call, and simply transitions back to s until data becomes available. Alternatively, some operators, such as Eddy [3] and XJoin [36] are designed to pursue other computation (e.g. transition to state other than s) when no data is available on a particular input; the Fjords programming model is naturally suited to such adaptive query processing systems.

Formulating query plan operators as state-machine states presents several interesting issues. It leads to operators which are neither "push" nor "pull": they simply look for input and operate on that input when it is available. Traditional pull-based database semantics are implemented via the queue between two operators: when an operator looks for data on a pull-based input queue, that queue issues a procedure call to the child

[1] This type of join is similar to the multiplex stream-combination operator in the Tribeca system [35], or the band-join proposed in [9].

```
public class Select extends Module {
    Predicate filter; //The selection predicate to apply
    QueueIF inputQueue;
    ...
    public TupleIF transition(StateIF state) {
        MsgIF mesg;
        TupleIF tuple = null;
        //Look for data on input queue
        mesg = inputQueue.get();
        if (mesg != null) {
            //If this is a tuple, check to see if it passes predicate
            if (mesg instanceof TupleMsg &&
                filter.apply(((TupleMsg)mesg).getTuple()))
                    tuple=((TupleMsg)mesg).getTuple();
            else ... handle other kinds of messages ...
        }
         ... adjust state: Select is stateless, so nothing to do here ...
         return tuple; //returning null means nothing to output
    }
}
```

(a)Selection Operator

```
public class PullQueue implements
QueueIF {
    //Modules this Queue connects
    Module below, above;
    StateIF bSt;
    ...
    public MsgIF get() {
        TupleIF tup=null;
        //Loop, pulling from below
        while (tup == null) {
            tup=below.transition(bSt);
            ... check for errors, idle ...
        }
        return new TupleMsg(tuple);
    }
}
```

(b)Pull Queue

Figure 5: *Code Snippet For Selection Operator and Pull Queue*

operator asking it to produce data and forces the caller to block until the child produces data. This allows operators to be combined in arbitrary arrangements of push and pull.

Figure 5 shows pseudocode for an example selection operator (Figure 5a) and pull queue (Figure 5b.) The selection operator simply checks its queue to see if there is data available; the queue may or may not actually return a tuple. If that queue is an instance of a pull queue, the *transition* method of the operator below will be called until it produces a tuple or an error.

State-machine programming requires a shift in the conventional thinking about how operators work. Telling a module to *transition* does not force it to produce a tuple, or constrain it to producing only a single tuple. The module computes for some period of time, then returns control to the scheduler which chooses another module to *transition* to. A badly behaved operator can block for long periods of time, effectively preventing other operators from being able to run if all modules are in a single thread. The scheduler can account for this to some extent by using a ticket scheme where operators are charged a number of tickets proportional to the amount of time they compute, as in [37]. This problem, however, is not restricted to Fjords: in general, faulty operators can impair an iterator-based query plan as well. In our experience, we have not found that this state-machine model makes operator development overly difficult.

One important advantage of a state machine model is that it reduces the number of threads. Traditional push-based schemes place each pushing operator in its own thread; that operator produces data as fast as possible and enqueues it. The problem with this approach is that operator-system threads packages often allow only very coarse control over thread scheduling, and database systems may want to prioritize particular operators at a fine granularity rather than devoting nearly-equal time slices to all operators. Our scheduler mechanism enables this kind of fine-grain prioritization by allowing Fjord builders to specify their own scheduler which *transitions* some modules more frequently than others. Furthermore, on some operating systems, threads are quite heavyweight: they have a high memory and context-switch overhead [39]. Since all state machine operators can run in a single thread, we never pay these penalties, regardless of the operating system.

3.1.4 Sensor-Sensitive Operators

Fjords can use standard database operators, but to be able to run queries over streaming data, special operators that are aware of the infinite nature of streams are required, such as those proposed in [33]. Some relational operators, like selections and projections, work with streams without modification. Others cannot be applied to a stream of data: aggregates like average and count and sorts fall under this category. Some join implementations, such as sort-merge join, which require the entire outer relation also fail. We use a variety of special operators in place of these solutions.

First, non-blocking join operators can be used to allow incremental joins over streams of data. Such operators have been discussed in detail in adaptive query processing systems such as Xjoin [36], Tukwila [20], Eddy [3] and Ripple Joins [14]. We have implemented an in memory symmetric hash-join [40], which maintains a hashtable for each relation. When a tuple arrives, it is hashed into the appropriate hash table, and the other relation's table is probed for matches. This technique finds all joining tuples, is reasonably efficient, is tolerant to delays in either data source, and is non-blocking with respect to both relations. Of course, when streams are infinite and storage is limited, eventually some data must be discarded. We chose to implement a "cleaner" process which scans hashtables for older tuples to discard when a fill-threshold is reached.

It is also possible to define aggregate operators, like count and average, which output results periodically; whenever a tuple arrives from the stream, the aggregate is updated, and its revised value is forwarded to the user. Of course, this incremental aggregation requires operators upstream from the aggregates to understand that values are being updated, and for the user interface to redisplay updated aggregate values. Similar techniques were also developed in the context of adaptive databases, for instance, the CONTROL Project's Online Aggregation algorithms [15] and the Niagara Internet Query System [34].

If traditional (i.e. blocking) aggregates, sorts, or joins must be used, a solution is to require that these operators specify a subset of the stream which they operate over. This subset is typically defined by upper and lower time bounds or by a sample count. Defining such a subset effectively converts an infinite stream into a regular relation which can be used in any database operator. This approach is similar to previous work done on windows in sequence database systems [33, 35].

Instead of using windowed operators, we can rely on a user interface to keep tuples in a sorted list and update aggregates as tuples arrive. Such an interface needs other functionality: as tuples stream endlessly in, it will eventually need to discard some of them, and thus some eviction policy is needed. Furthermore, since queries never end, the user needs to be given a way to stop a continuous query midstream. The Control project discusses a variety approaches to such interfaces[16].

By integrating these non-blocking operators into our system, we can take full advantage of Fjords' ability to mix push and pull semantics within a query plan. Sensor data can flow into Fjords, be filtered or joined by non-blocking operators, or be combined with local sources via windowed and traditional operators in a very flexible way.

3.2 Sensor Proxy

The second major component of our sensor-query solution is the sensor-proxy, which acts as an interface between a single sensor and the Fjords querying that sensor. The proxy serves a number of purposes. The most

important of these is to shield the sensor from having to deliver data to hundreds of interested end-users. It accepts and services queries on behalf on the sensor, using the sensor's processor to simplify this task when possible.

In order to keep the sensor's radio-receiver powered down as much as possible, the proxy only sends control messages to the sensor during limited intervals. As a sensor relays samples to the proxy, it occasionally piggybacks a few bytes indicating that it will listen to the radio for some short period of time following the current sample. The exact duration of this window and interval between windows depends on the sensors and sample rates in question; we expect that for our traffic sensor environment the window size will be a few tens-of-milliseconds per second. Though this is a significant power cost, it is far less than the cost of keeping the receiver powered up at all times.

One function of control messages is to adjust the sample rate of the sensors, based on user demand. If users are only interested in a few samples per second, there's no reason for sensors to sample at hundreds of hertz, since lower sample rates are directly proportional to longer battery life. Similarly, if there are no user queries over a sensor, the sensor proxy can ask the sensor to power off for a long period, coming online every few seconds to see if queries have been issued.

An additional role of the proxy is to direct the sensor to aggregate samples in predefined ways, or to download a completely new program into the sensor if needed. For instance, in our traffic scenario, the proxy can direct the sensor to use one of several sampling algorithms depending on the amount of detail user-queries require. Or, if the proxy observes that all of the current user queries are interested only in samples with values above or below some threshold, the proxy can instruct the sensor to not transmit samples outside that threshold, thereby saving communication.

Sensor-proxies are long-running services that exist across many user queries and route tuples to different query operators based on sample rates and filtration predicates specified by each query. When a new user query over a sensor-stream is created, the proxy for that sensor is located and the query is installed. When the user stops the query, the proxy stops relaying tuples for that query, but continues to monitor and manage the sensor, even when no queries are being run.

We expect that in many cases there will be a number of users interested in data from a single sensor. As we show in Section 4.3 below, the sensor proxy can dramatically increase the throughput of a Fjord by limiting the number of copies of sensor tuples flowing through the query processor to just one per sample, and having the user queries share the same tuple data.

3.3 Building A Fjord

Given Fjord operators and sensor proxies as the main elements of our solution, it is straightforward to generate a Fjord from a user query over a sensor. For this discussion, we will make the simple assumption that queries consist of a set of selections to be applied, a list of relations to be joined, and an optional aggregation and grouping expression. We are not focused on a particular query language, and believe Fjords are a useful architectural construct for any query language – other research projects, such as Predator and Tribeca, have proposed useful languages for querying streaming data, which we believe are readily adaptable to our architecture [33, 35]. We

do not allow joins of two streams [2] nor can we aggregate or sort a stream. Users are allowed to define windows on streams which can be sorted, aggregated, or joined. A single stream can be joined with a stream-window or a fixed data source if it is treated as the outer relation of an index join or the probe relation of a hash-join. Users can optionally specify a sample-rate for sensors, which is used to determine the rate at which tuples are delivered for the query.

Building the Fjord from such a query works as follows: for each base relation r, if r is a sensor, we locate the persistently running sensor proxy for r. We then install our query into r's proxy, asking it to deliver tuples at the user-provided sample rate and to apply any filters or aggregates which the user has specified for the sensor-stream. The proxy may choose to fold those filters or aggregates into existing predicates it has been asked to apply, or it may request that they be managed by separate operators. For those relations r' that do not represent sensors, we create a new scan operator over r'. We then instantiate each selection operator, connecting it to a base-relation scan or earlier selection operator as appropriate. If the base-relation is a sensor, we connect the selection via a push-queue, meaning that the sensor will push results into the selection. For non-sensor relations, we use a pull queue, which will cause the selection to invoke the scan when it looks for a tuple on its input queue.

We then connect join-operators to these chains of scans and selects, performing joins in the order indicated by a standard static query optimizer, such as the one presented in [32]. If neither of the joined relations represents a sensor, we choose the join-method recommended by the optimizer. If one relation is a sensor, we use it as the probe relation of a hash join, hashing into the static relation as each stream tuple is pushed into the join. The output of a join is a push queue if one relation is from a sensor, and a pull queue otherwise.

Sorts and aggregates are placed at the top of the query plan. In the case where one of the relations is from a sensor, but the user has specified a window size, we treat this as a non-sensor relation by interposing a filtration operator above the sensor proxy which passes only those tuples in the specified window.

3.4 Multiple Queries in a Single Fjord

One way in which streaming data differs from traditional data sources is that it is inseparably tied with the notion of *now*. Queries over streams begin looking at the tuples produced starting at the instant the query is posed – the history of the stream is not relevant. For this reason, it is possible to share significant amounts of computation and memory between several queries over the same set of data sources: when a tuple is allocated from a particular data source, that tuple can immediately be routed to all queries over that source – effectively, all queries are reading from the same location in the streaming data set. This means that streaming tuples need only be allocated once, and that selection operators over the same source can apply multiple predicates at once. Fjords explicitly enable this sharing by instantiating streaming scan operators with multiple outputs that allocate only a single copy of every streaming tuple; new queries over the same streaming source are folded into an existing Fjord rather than being placed in a separate Fjord. A complete discussion of how this allocation and query folding works is beyond the scope of this paper, but related ideas are presented in literature on continuous queries [6, 23]. We will, however, show how this folding can be used to improve query performance in the results section which follows.

[2] Two streams can be joined by "zippering" them together as they flow past, but this works only when streams are synchronized and eqi-joined on sample time, as in Figure 4 above.

4 Traffic Implementation and Results

We now present two performance studies to motivate the architecture given above. The first study, given in this section, covers the performance of Fjords. The second study, given in Section 5, examines the interaction of sensor power consumption and the sensor-proxy and demonstrates several approaches to traffic sensor programs which can dramatically alter sensor lifetime.

In this section, we show how the Fjord and sensor-proxy architectures can be applied to the traffic-sensor environment. We begin by giving examples of queries over those sensors, then show alternatives for translating one of those queries into a Fjord, and then finally present two Fjord performance experiments.

4.1 Traffic Queries

We present here two sample queries which we will refer to through the rest of the section, as given by the following SQL excerpts. These queries are representative of the types of queries commuters might realistically ask of a traffic inquiry system. They are not meant to test the full functionality of our query evaluation engine. We are not, as of yet, committed to SQL as the language of choice for queries over sensor data but it is the language our tools currently support.

These queries are run over data from 32 of CalTrans' inductive loops collected by an older-generation of sensors equipped with wireless radio links that relay data back to UC Berkeley. These sensors consist of sixteen sets of two sensors (referred to as "upstream" and "downstream"), with one pair on either side of the freeway on eight distinct segments of I-80 near UC Berkeley. The sensors are 386-class devices with Ricochet 19.2 kilobit modem links to the Internet. They collect data at 60Hz and relay it back to a Berkeley server, where it is aggregated into counts of cars and average speeds or distributed to various database sources (such as ours) via JDBC updates. Because we cannot actually deploy a mote-based sensor network onto the freeway, we use this streaming data for the Fjord-experiments presented in this section.

Query 1

```
SELECT AVG(s.speed, w)
FROM sensorReadings AS s
WHERE s.segment ∈
knownSegments
```

This query selects the average speed over segments of the road, using an average window interval of w. These queries can be evaluated using just the streaming data currently arriving into the system.They require no additional data sources or access to historical information.

Query 2

```
SELECT AVG(s.speed, w), i.description
FROM incidents as i,
  sensorReadings as s
WHERE i.time >= now - timeWindow
GROUP BY i.description
HAVING speedThreshold >
       (SELECT AVG(s.speed,w)
        FROM sensorReadings as s
        WHERE i.segment = s.segment
        AND s.segment ∈ {knownSegments})
```

This query joins sensor readings from slow road segments the user is interested in to traffic incidents which

are known to have recently occurred in the Bay Area. Slow road segments are those with an average speed less than $speedThreshold$. The set of segments the user is interested in is $knownSegments$. Recently means since $timeWindow$ seconds before the current time. The California Highway Patrol maintains a web site of reported incidents all over California, which we can use to build the incidents relation. [5]. Evaluating this query requires a join between historical and streaming data, and is considerably more complicated to evaluate than Query 1.

4.2 Traffic Fjords

In this section, we show two alternative Fjords which correspond to Query 1 above. Space limitations preclude us from including similar diagrams for Query 2; we will discuss the performance of Query 2 briefly in Section 4.3.2.

Figure 6 shows one Fjord which corresponds to Query 1. Like the query, it is quite simple: tuples are routed first from the BHL server to a sensor proxy operator, which uses a JDBC input queue to accept incoming tuples. This proxy collects streaming data for various stations, averages the speeds over some time interval, and then routes those aggregates to the multiplex operator, which forwards tuples to both a save-to-disk operator and a filter operator for the $\in$ clause in query 1. The save-to-disk operator acts as a logging mechanism: users may later wish to recall historical information over which they previously posed queries. The filter operator selects tuples based on the user query, and delivers to the user a stream of current speeds for the relevant road segment.

Notice that the data flow in this query is completely push driven: as data arrives from the sensors, it flows through the system. User queries are used to parameterize Fjord operators, but the queries themselves do not cause any data to be fetched or delivered. Also note that user queries are continuous: data is delivered periodically until the user aborts the query. The fact that data is pushed from sensors eliminates problems that such a system could experience as a result of delayed or missing sensor data: since the sensor is driving the flow of tuples, no data will be output for offline sensors, but data from other sensors flowing through the same Fjord will not be blocked while the query processor waits for those offline sources.

Figure 6 works well for a single query, but what about the case where multiple users pose queries of the same type as Query 1, but with different filter predicates for the segments of interest? The naive approach would be to generate multiple Fjords, one per query, each of which aggregates and filters the data independently. This is clearly a bad idea, as the allocation and aggregation of tuples performed in the query is identical in each case. The ability to dynamically combine such queries is a key aspect of the Fjords architecture, and is similar to work done as a part of Wisconsin's NiagaraCQ project[6]. A Fjord with such a combined sensor-proxy is illustrated in Figure 7.

4.3 Fjords for Performance

Having described the Fjords for our example queries, we now present two experiments related to Fjords: In the first, we demonstrate the performance advantage of combining related queries into a single Fjord. In the second, we demonstrate that the Fjords architecture allows us to scale to a large number of simultaneous queries.

We implemented the described Fjords architecture, using join and selection operators which had already been built as a part of the Telegraph dataflow project. All queries were run on a single, unloaded Pentium III 933Mhz with a single EIDE drive running Linux 2.2.18 using Sun's Java Hotspot 1.3 JDK. To drive the experiments we use traces obtained from the BHL traffic sensors. These traces are stored in a file, which is read

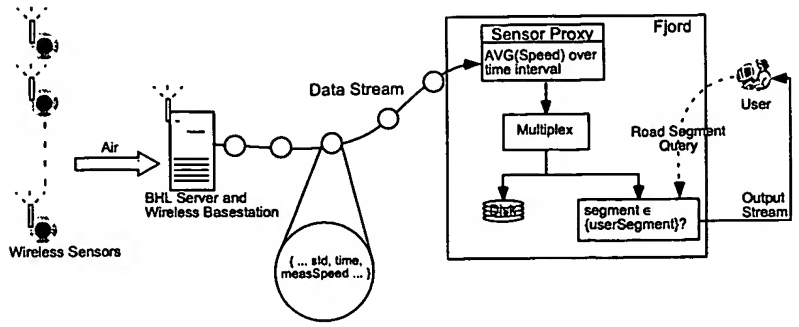


Figure 6: *Fjord Corresponding to Query 1*

once into a buffer at the beginning of each experiment so that tests with multiple queries over a single sensor are not penalized for multiple simultaneous disk IOs on a single machine. This data set consists of records of the form $\{time, lane, station, avg(speed), flow\}$, with the following values:

$time$:	The start time of this sample window
$lane$:	The lane for this sample
$station$:	The station number for this sample
$avg(speed)$:	The average speed of vehicles observed in this sample
$flow$:	The number of vehicles observed in this sample

For the particular queries discussed here, sample window size is not important, so we generate traces with 30-second windows. The trace file contained 10767 30-byte records corresponding to traffic flow at a single sensor during June '00.

4.3.1 Combining Fjords to Increase Throughput

For the first experiment, we compare two approaches to running multiple queries over a single streaming data source. For both approaches, some set of n user queries, Q, is submitted. Each query consists of a predicate to be evaluated against the data streaming from a single sensor. The first approach, called the *multi-Fjord* approach

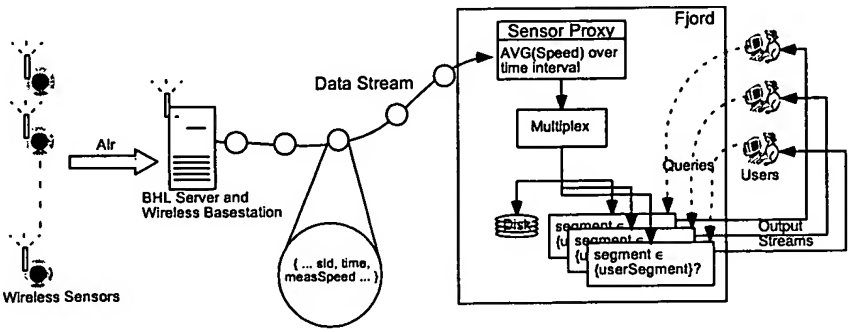


Figure 7: *Fjord Corresponding to Query 1 For Multiple User Queries*

allocates a separate Fjord (such as the one shown in Figure 6) for each query $q \in Q$. In the second approach, called the *single Fjord* approach, just one Fjord is created for all of the queries. This Fjord contains a filter operator for each of the n queries (as shown in Figure 7.) Thus, in the first case, n threads are created, each running a Fjord with a single filter operator, while in the second case, only a single thread is running, but the Fjord has n filter operators. We implemented a very simple round-robin scheduler which schedules each operator in succession, one after the other, which is used in both cases.

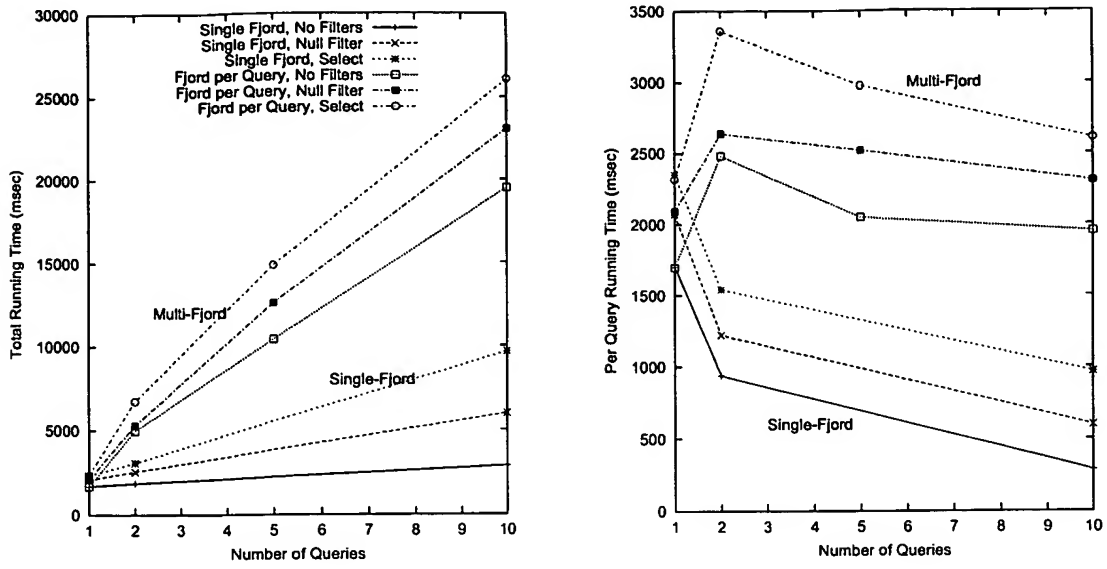
In order to isolate the cost of evaluating filters, we also present results for both of these architectures when used with no filter operator (e.g. the sensor proxy outputs directly to the user-queue), and with a null-operator that simply forwards tuples from the sensor proxy to the end user.

Figure 8 shows the performance results of the two experiments, showing total query time (Figure 8(a)) and time per query (Figure 8(b)). All experiments were run with 150 MB of RAM allocated to the JVM and with a 4MB tuple pool allocated to each Fjord. Notice that the single-Fjord version handily outperforms the multi-Fjord version in all cases, but that the cost of the selection and null-filter is the same in both cases (300 and 600 milliseconds per query, respectively). This behavior is due to several reasons: First, there is substantial cost for laying out additional tuples in the buffer pools of each of the Fjords in the multi-Fjord case. In the single Fjord case, each tuple is read once from disk, placed in the buffer pool, and never again copied. Second, there is some overhead due to context switching between multiple Fjord-threads. Figure 8(b) reflects the direct benefit of sharing the sensor-proxy: additional queries in the single-fjord version are less expensive than the first query, whereas they continue to be about the same amount of work as a single query in the multi-fjord version. The spike in the multi-fjords lines at two queries in 8(b) is due to queries building up very long queues of output tuples, which are drained by a separate thread. Our queues become slower when there are more than a few thousand elements on them. This does not occur for more simultaneous queries because each Fjord-thread runs for less time, and thus each output queue is shorter and performs better. This is the same reason the slope of the single-fjord lines in Figure 8(b) drops off: all queries share a single output queue, which becomes very long for lots of queries.

4.3.2 Scaling to a Large Number of Queries

In the previous section, we showed the advantage of combining similar user-queries into a single Fjord. Now, in the second experiment, we show that this solution makes it possible to handle a very large number of user queries. In these tests, we created n user queries, each of which applied a simple filter to the same sensor stream, in the style of Query 1 in Section 4.1. We instantiated a Fjord with a single sensor proxy, plus one selection operator per query. We allocated 150MB of RAM to the query engine and gave the Fjord a 4MB tuple pool. We used the same data file as in the previous section. Figure 9(b) shows the number of tuples which the system processes per second per query versus n, while Figure 9(a) shows the the aggregate number of tuples processed versus n. The number of tuples per second per query is the limit of the rate at which sensors can deliver tuples to all users and still stay ahead of processing. Notice that total tuple throughput climbs up to about 20 queries, and then remains fairly constant. This leveling off happens as the processor load becomes maximized due to evaluation of the select clauses and enqueuing and dequeuing of tuples.

We also ran similar experiments from Query 2 (Section 4.1). Due to space limitations, we do not present these results in detail. The results of this experiments were similar to the Query 1 results: the sensor-proxy



(a)Total Running Time vs. No. of Queries (b) Time Per Query vs. No. of Queries

Figure 8: *Effect of Combining Multiple Queries Into A Single Fjord*

amortizes the cost of stream-buffering and tuple allocation across all the queries. With Query 2, the cost of the join is sufficiently high that the benefit of this amortization is less dramatic: 50 simultaneous queries have a per query cost which is only seven-percent less than the cost of a single query.

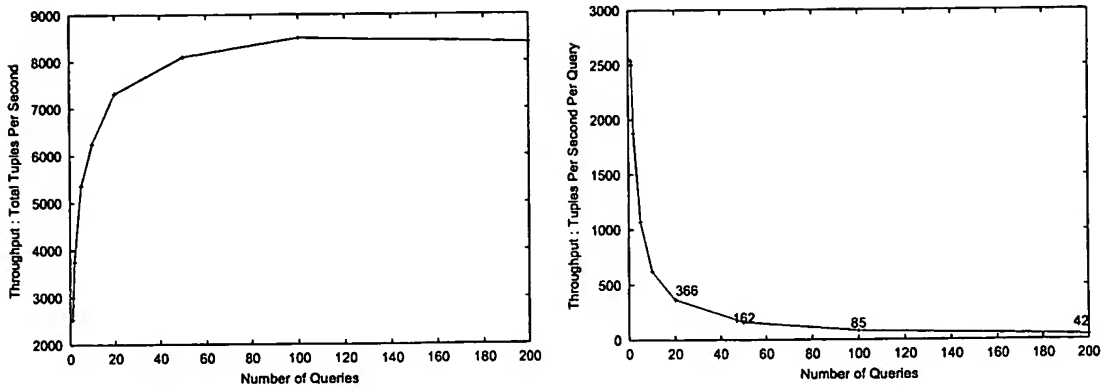
5 Controlling Power Consumption via Proxies

The experiments in the previous section demonstrated the ability of Fjords to efficiently handle many queries over streaming data. We now turn our attention to another key aspect of sensor query processing, the impact of sample rate on both sensor lifetime and the ability of Fjords to process sensor data. In this section, we focus on sensors that are similar to the wireless sensors motes described in Section 2.1.2 above. We choose to use this design rather than the fixed traffic sensors which are currently deployed on the freeway because we believe this is a more realistic model of the way sensors in the near future will behave, and because SmartDust sensors can be remotely reprogrammed, which is necessary for implementing the techniques we describe below.

5.1 Motes for Traffic

In this section, we assume that motes are placed or can self-organize into pairs of sensors less than a car's-length apart and in the same lane. We call these sensors S_u, the upstream sensor, and S_d, the downstream sensor. We assume that through radio-telemetry with fixed basestations and each other, of the sort described in [28], it is possible for the sensors to determine that their separation along the axis of the road is d feet. These sensors are equipped with light or infrared detectors that tell them when a car is passing overhead.

Traffic engineers are interested in deducing the speed and length of vehicles traveling down the freeway; this is done via four time-readings: t_0, the time the vehicle covers S_u; t_1, the time the vehicle completely covers



(a) Total Tuples Per Second vs. No. of Queries

(b) Tuples Per Second Per Query vs. No. of Queries

Figure 9: *Tuple Throughput vs. Number of Queries*

both S_u and S_d ; t_2, the time the vehicle ceases to cover S_u, and t_3, the time the vehicle no longer covers either sensor. These states are shown in Figure 10. Notice that the collection of these times can be done independently by the sensors, if the query processor knows how they are placed: S_u collects t_0 and t_2, while S_d collects t_1 and t_3. Given that the sensors are d feet apart, the speed of a vehicle is then $d/(t_1 - t_0) ft/sec$, since $t_1 - t_0$ is the amount of time it takes for the front of the vehicle to travel from one sensor to the other to the other. The length of the vehicle is just $speed \cdot (t_0 - t_2)$, since $t_0 - t_2$ is the time it takes for both the front and back of the car to pass the S_u[3].

These values are important because they indicate how accurate the timing measurements from the sensors need to be; imagine for instance that we wanted an estimate of the length of a car to an accuracy of one foot. To do this, readings t_0 and t_2 must be made to an accuracy of 6 inches, so that the maximum error of $t_0 - t_2$ is no greater than 12 inches. Thus, the car cannot travel more than 6 inches between sample periods. At 60mph, simple physics tells us a car will travel 6 inches in .0056 seconds, requiring a sample rate of about 180Hz. This is a relatively high sample rate, and, as we will show in the next section, in a naive implementation where sensors transmit every sample back to the host computer for speed and length detection, is high enough to severely limit the life of sensors and constrain the performance of the Fjord. Notice that this calculation assumes the speed of the vehicle is known; if we are estimating the length of the vehicle from our speed estimation, an even faster sample rate is needed.

Sensors relay readings to base stations placed regularly along the freeway. These base stations have wired Internet connections or high-power wireless radios which can relay information back to central servers for processing. Such base stations are elevated, to overcome propagation problems that result between sensors that are on or embedded in a low-reflectivity surface like the roadway. Sensors transmit samples along with time-stamps, which can be generated via techniques such as those proposed by researchers at UCLA [10] where base-stations periodically broadcast network-time (NTP) values to the sensors, which the sensors use to adjust local skew in their clocks. Time stamps are used to coordinate readings from sensor pairs; the NTP based scheme

[3]This analysis was derived based on computations for inductive loop sensors in [7].

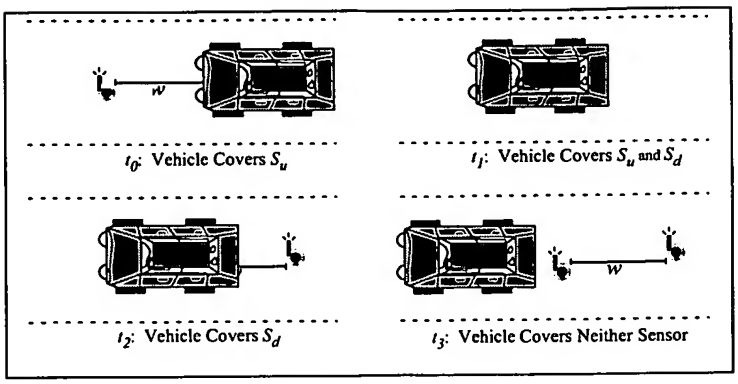


Figure 10: *Vehicle moving across sensors S_u and S_l at times t_0,t_1,t_2, and t_3*

proposed above can keep sensor clocks synchronized within $1uS$ of each other. Since a 60mph car will travel only 2mm in a microsecond, this margin of error is more than acceptable to produce measurements accurate to 1 foot, our desired accuracy for the vehicle length computations shown above.

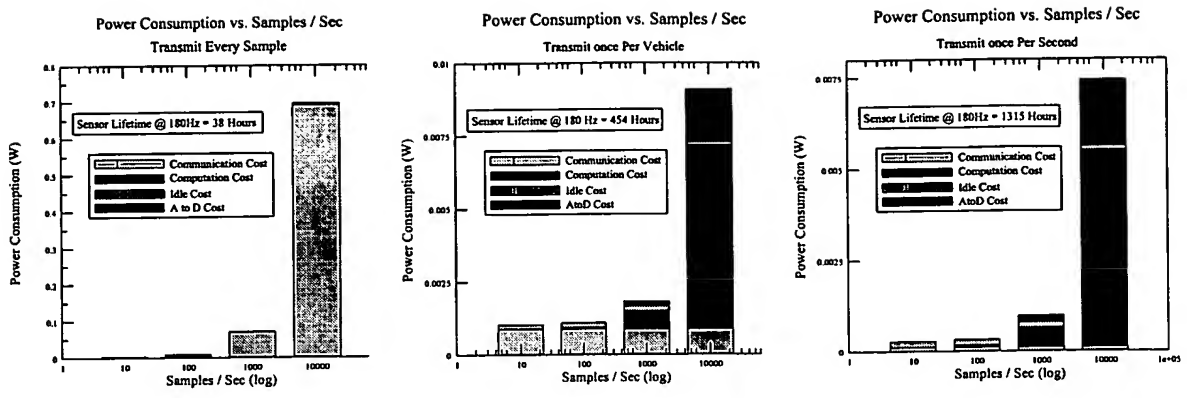
5.2 Sensor Power-Consumption Simulation

The results described in this section were produced via simulation. Processor counts were obtained by implementing the described algorithms on an Atmel simulator, power consumption figures were drawn from the Atmel 8515 datasheet [2], and communication costs were drawn from the TinyOS results in [18], which uses the RFM TR100 916 Mhz [29] radio transceiver. Table 1 summarizes the communication and processor costs used to model power consumption in this section.

We present three sensor scenarios. In the first, sensors relay data back to the host PC at their sample rate, performing no aggregation or processing, and transmitting raw voltages. The code is extremely simple: the sensor reads from its A-to-D input, uses the TinyOS communications protocol to transmit the sample, then sleeps until the next sample period arrives. In this naive approach, power consumption is dominated by communication costs. Figure 11(a) illustrates this; the idle cost, computation cost, and A-to-D costs are all so small as to be nearly invisible. For the baseline sample rate of 180Hz, the power draw comes to 13mW or 2.6mA/h, enough for our sensor pairs to power themselves for about a day and a half: clearly this approach does not produce low-maintenance road sensors. Furthermore, this approach places a burden on the database system: as Figure

Table 1: **Sensor Parameter Values.** *Power Parameters for Atmel 8515 Processor and RFM TR100 Radio.*

Parameter	Value
Radio Xmit Energy Cost	$4.31^{-6}\ J/bit$
Processor Voltage	$5V$
Processor Current (Active)	$6\,mW$
Processor Current (Idle)	$30\,\mu W$
Processor Speed	$1\,Mhz$
A-to-D Current	$.6\,mW$
A-to-D Latch Time	$60\,\mu S$
Battery Capacity	$100\,mAh$



(a) (b) (c)

Figure 11: *Power consumption for different sensor implementations*

9(a) shows, at 180 samples/per second a Fjord is limited to about 50 simultaneous simple queries, if the entire sample stream is routed through each query. In practice, of course, not all of the queries are interested in the entire data stream, so the sensor-proxy can aggregate the samples into identified vehicles or vehicle counts.

In the second scenario (shown in Figure 9(b)), we use a small amount of processor time to dramatically reduce communication costs. Instead of relaying raw voltages, the sensors observe when a car passes over them, and transmit the $\{t_0, t_2\}$ or $\{t_1, t_3\}$ tuples needed for the host computer to reconstruct the speed and length of the vehicle. The sensors still sample internally at a fast sample rate, but relay only a few samples per second – in this case, we assume no more than five vehicle pass in any particular second. This corresponds to a vehicle every 18 feet at 60 MPH, which is a very crowded (and dangerous!) highway. Power consumption versus sample rate on each sensor is shown in Figure 11(b). In this example, for higher sample rates, power consumption is dominated by the processor and A-to-D converter; communication is nearly negligible. At 180Hz, the total power draw has fallen to 1.1mW, or .22mA/h, still not ideal for a long lived sensor, but enough to power our traffic sensors for a more reasonable two and a half weeks. Also, by aggregating and decreasing the rate at which samples are fed into the query processor, the sensors contribute to the processing of the query and require fewer tuples to be routed through Fjords. Although this may seem like a trivial savings in computation for a single sensor, in an environment with hundreds or thousands of traffic sensors, it is non-negligible.

In the final scenario, we further reduce the power demands by no longer transmitting a sample per car. Instead, we only relay a count of the number of vehicles that passed in the previous second, bringing communications costs down further for only a small additional number of processor instructions per sample. This is shown in Figure 11(c); the power draw at 180Hz is now only .38mW, a threefold reduction over the second scenario and nearly two orders of magnitude better than the naive approach. Notice that the length and speed of vehicles can no longer be reconstructed; only the number of vehicles passing over each sensor per second is given. We present this scenario as an example of a technique that a properly programmed sensor-proxy could initiate when it determines that all current user queries are interested only in vehicle counts.

To summarize, for this particular sensor application, there are several possible approaches to sampling sensors. For traffic sensors, we gave three simple sampling alternatives which varied in power consumption by nearly two orders of magnitude. Thus, the choice of sensor programs reflect essential tradeoffs when dealing

with low-power computing, and have a dramatic effect on battery life. Lowering the sample rate increases sensor lifetime but reduces the accuracy of the sensor's model. Aggregating multiple samples in memory increases processor and CPU burden but reduces communication cost. Therefore, a sensor-proxy which can actively monitor sensors, weighing user needs and current power conditions, and appropriately program and control sensors is necessary for getting acceptable sensor battery life and performance.

6 Related Work

Having presented our solution for queries over stream sensor data, we now discuss related projects in the sensor and database domains.

6.1 Related Database Projects

The work most closely related to ours is the Cougar project at Cornell [26]. Cougar is also intended for query processing over sensors. Their research, however, is more focused on modeling streams as persistent, virtual relations and enabling user queries over sensors via abstract data-types. Their published work to date does not focus on the power or resource limitations of sensors, because it has been geared toward larger, powered sensors. They do not discuss the push based nature of sensor streams. Our work is complementary to their in the sense that they are more focused on modeling sensor-streams, whereas we are interested in the practical issues involved in efficiently running queries over streams.

There has been recent work on databases to track moving objects, such as [41]. In such systems, object position updates are similar to sensor tuples in that they arrive unpredictably and their arrival rate is directly tied to number of queries which the database system can handle. Thus, such mobile databases could be an application for the techniques we present in this paper.

6.2 Distributed Sensor Networks

In the remote sensing community, there are a number of systems and architecture projects focused on building sensor networks where data-processing is performed in a distributed fashion by the sensors themselves. In these scenarios, sensors are programmed to be application-aware, and operate by forwarding their readings to nearby sensors and collecting incoming readings to produce a locally consistent view of what is happening around them. As information propagates around the network, each sensor is better able to form a model of what is going on. An example of such a project is the Directed Diffusion model from the USC Information Sciences Institute; Users pose queries to a particular sensor about people or vehicles moving in some region, and sensors near that region propagate their local readings to the queried which it aggregates to answer the query [19]. It is exciting to think that such a decentralized model of query processing is possible; however, we believe that, at least for the current generation of sensors, power, storage, and processor limitations dictate that fixed, powered servers are needed to perform large scale sensor data-processing on the sensors themselves.

6.3 Data Mining on Sensors

Current scientific sensor research is similar to any data mining and warehousing application: data flows into local servers from each sensor, which create on-disk sensor logs. At some later point, this data is shipped to a central repository, which scientists then connect to and execute large aggregate and subset queries to facilitate their data exploration. The DataCutter [4] system exemplifies current research in this area: it focuses on providing an efficient platform in which scientists can collect data from multiple repositories and then efficiently aggregate and subset that data across multiple dimensions. This model is very different from the interactive sensor processing environment we envision.

6.4 Languages and Operators for Streams

There has been significant work in the database community focused on the language and operator design issues in querying streams. Early work sought to design operators for streams in the context of functional programming languages like Lisp and ML [25], or for specialized regimes like network-router log analysis [35]. Seshadri, et. al. [33] bring this work fully into the domain of relational databases by describing extensions to SQL for stream-query processing via windowed and incremental operators.

The CONTROL project [16] discusses the possibility of user interfaces for the incrementally sorting and aggregating very large data sets which is also applicable to streams. Shanmugasundaram et. al, [34] discuss techniques for percolating partial aggregates to end users which also apply.

More recent research on streams continues to extend relational databases with complex operators for combining and mining data streams. For instance, [12] showed single pass algorithms to compute complex, correlated aggregates over sets of streams.

6.5 Continuous Queries

Existing work on continuous queries provides some interesting techniques for simultaneously processing many queries over a variety of data sources. These systems provide an important starting point for our work but are not directly applicable as they are focused on continuous queries over traditional database sources or web sites and thus don't deal with issues specific to streaming sensor data.

In the OpenCQ system [23], continuous queries are very similar to SQL triggers, except that they are more general with respect to the tables and columns they can be defined over and are geared toward re-evaluation of expressions as data sources change rather than dependency maintenance. In OpenCQ, continuous queries are four-element tuples consisting of a SQL-style query, a trigger-condition, a start-condition, and an end-condition. Some effort is taken to combine related predicates to eliminate redundant evaluations, but the NiagaraCQ system, discussed next, presents a more robust solution to this problem.

The NiagaraCQ project [6] is also focused on providing continuous queries over changing web sites. Users install queries, which consist of an XML-QL [8] query as well as a duration and re-evaluation interval. Queries are evaluated periodically based on whether the sites they query have changed since the query was last run. The system is geared toward running very large numbers of queries over diverse data sources. The system is able to perform well by grouping similar queries, extracting the common portion of those queries, and then evaluating the common portion only once. We expect that this technique will apply to streaming sensor queries as well:

there will be many queries over a particular stream which share common subexpressions.

7 Future Work and Conclusions

As wireless communications technology and embedded processing technologies converge to allow the large scale deployment of tiny, low-power, radio-driven sensors, efficient, power sensitive techniques for querying the data they collect will be crucial. Other research has addressed languages and operators for stream processing, but ignored issues of data-flow integration and sensor power-management.

Our sensor-stream processing solution addresses the low-level infrastructure issues in a sensor-stream query processor via two techniques: First, the Fjords architecture combines proxies, non-blocking operators and conventional query plans. This combination allows streaming data to be pushed through operators that pull from traditional data sources, efficiently merging streams and local data as samples flow past. Second, sensor-proxies serve as mediators between sensors and query plans, using sensors to facilitate query processing while being sensitive to their power, processor, and communications limitations.

There are a number of interesting areas for future work. In particular, it would be useful to integrate tools for combining common sub-parts of user queries. We believe that many users will formulate similar sensor queries, and tools for rapidly evaluating similar classes of queries are needed. The XFilter[1] and NiagaraCQ[6] projects both offer techniques for extracting and evaluating common parts of user queries.

Secondly, Fjords are currently non-adaptive; that is, they do not modify the order of joins in the face of sensor delays or intermittent failures. We plan to explore the use of adaptive query operators such as Eddy[3] and XJoin[36] in Fjords. We believe that these operators can serve an important role in integrating non-streaming and streaming data by buffering streaming sources and masking latencies in traditional sources.

The solutions we have presented are an important part of the Telegraph Query Processing System, which seeks to extend traditional query processing capabilities to a variety of non-traditional data sources. Our sensor-stream processing techniques allow Telegraph to query sensors seamlessly and efficiently.

References

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *International Conference on Very Large Data Bases*, September 2000.

[2] Atmel Corporation. Atmel 8bit AVR microcontroller datasheet. http://www.atmel.com/atmel/acrobat/doc1041.pdf.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[4] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 2000 Mass Storage Systems Conference*. IEEE Computer Society Press, March 2000.

[5] California Highway Patrol. Traffic incident information page. http://cad.chp.ca.gov/.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD*, 2000.

[7] B. Coifman. Vehicle reidentification and travel time measurement using the existing loop detector infrastructure. In *Transportation Research Board*, 1998.

[8] A. Deutsch, M. Fernandez, D. Floresc, A. L. , and D. Suciu. XML-QL: A query language for XML, 1998. http://www.w3.org/TR/NOTE-xml-ql.

[9] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th Conference on Very Large Databases*, Barcelona, Spain, 1991.

[10] J. Elson and D. Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing*, April 2001.

[11] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, Salt Lake City, Utah, May 2001.

[12] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Santa Barbara, CA, May 2001.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[14] P. Hass and J. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD*, pages 287–298, Philadelphia, PA, June 1999.

[15] J. Hellerstein, P. Hass, and H. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD*, pages 171–182, Tucson, AZ, May 1997.

[16] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, 32(8), August 1999.

[17] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, N. Lanham, S. Madden, V. Raman, F. Reiss, and M. Shah. The design of telegraph: An adaptive dataflow system for streams of networked data. Submitted for Publication.

[18] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[19] C. Intanagonwiwat, R. Govindan, , and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, Boston, MA, August 2000.

[20] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD*, 1999.

[21] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM))*, Seattle, WA, August 1999.

[22] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th Annual IEEE/Mobicom Conference*, Seattle, WA, 1999.

[23] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[24] R. Min, M. Bhardwaj, S.-H. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. An architecture for a power-aware distributed microsensor node. In *IEEE Workshop on Signal Processing Systems (SiPS '00)*, October 2000.

[25] D. S. Parker. Stream databases. Technical report, UCLA, 1989. Final Report for NSF Grant IRI 89-17907.

[26] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management, Hong Kong*, January 2001.

[27] G. Pottie and W. Kaiser. Wireless sensor networks. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[28] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Proc. of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM)*, August 2000.

[29] RFM Corporation. RFM TR1000 Datasheet. http://www.rfm.com/products/data/tr1000.pdf.

[30] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. Santa Barbara, CA, May 2001.

[31] P. Saffo. Sensors: The next wave of innovation. *Communications of The ACM*, 40(2):92 – 97, February 1997.

[32] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. pages 23–34, Boston, MA, 1979.

[33] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database systems. In *International Conference on Very Large Data Bases*, Mumbai, India, September 1996.

[34] J. Shanmugasundaram, K. Tufte, D. DeWitt, J. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *Workshop on the Web and Databases (WebDB)*, May 2000.

[35] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[36] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000 2000.

[37] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 1–12, Monterey CA (USA), 1994.

[38] M. Weiser. Some computer science problems in ubiquitous computing. *Communications of the ACM*, July 1993.

[39] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley, April 2000. http://www.cs.berkeley.edu/ mdw/papers/events.pdf.

[40] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS*, pages 68–77, December 1991.

[41] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases fOr MovINg Objects tracking. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 547–549, Philadelphia, PA, June 1999. ACM Press.

Monitoring Streams-A New Class of DBMS Applications

Don Carney, Ugur Cetintemel, Mitch Cherniack,
Sangdon Lee,Greg Seidman, Michael Stonebraker,
Nesime Tatbul, Stan Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-02-01
February 2002

Monitoring Streams – A New Class of DBMS Applications

Don Carney
Brown University
dpc@cs.brown.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Sangdon Lee
Brown University
sdlee@cs.brown.edu

Greg Seidman
Brown University
gss@cs.brown.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Abstract

This paper introduces monitoring applications, which we will show differ substantially from conventional business data processing. The fact that a software system must process and react to continual inputs from many sources (e.g., sensors) rather than from human operators requires one to rethink the fundamental architecture of a DBMS for this application area. In this paper, we present the architecture of a new DBMS that is currently under construction at Brown, Brandeis, and M.I.T. We describe the basic system architecture, a stream-oriented set of operators, optimization tactics, support for real-time operation, and the design-time environment.

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these kinds of applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions to this repository. We call this a *Human-Active, DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. In addition, there is typically little or no support for querying a "trigger base." Fourth, DBMSs assume that data elements have precise values (e.g., employee salaries), and have little or no support for data elements that are imprecise or out-of-date. Lastly, DBMSs assume that applications require no real-time services.

There is a substantial class of applications where all five assumptions are problematic. Monitoring applications are applications that monitor continuous streams of data. This class of applications includes military applications that monitor readings from sensors worn by soldiers (e.g., blood pressure, heart rate, position), financial analysis applications that monitor streams of stock data reported from various stock exchanges, and tracking applications that monitor the locations of large numbers of objects for which they are responsible (e.g., audio-visual departments that must monitor the location of borrowed equipment). Because of the high volume of data monitored, and querying requirements for these applications, monitoring applications would benefit from DBMS support. Existing DBMS systems, however, are ill suited for such applications since they target business applications.

First, monitoring applications get their data from external sources (e.g., sensors) rather than from humans issuing transactions. The role of the DBMS in this context is to alert humans when abnormal activity is detected. This is a *DBMS-Active, Human-Passive* (*DAHP*) model.

Second, monitoring applications require data management extending over the entire history of values reported in a stream, and not just over the most recently reported values. Consider a monitoring application that tracks the location of items of interest, such as overhead transparency projectors and laptop computers, using electronic property stickers attached to the objects. Data is generated by ceiling-mounted sensors inside a building and the GPS system in the open air. If a reserved overhead projector is not in its proper location, then one might want to know the geographic position of the missing projector. In this case, the last value of the monitored object is required. However, an administrator might also want to know the duty cycle of the projector, thereby requiring access to the entire historical time series.

Third, most monitoring applications are trigger-oriented. If one is monitoring a chemical plant, then one wants to alert an operator if a sensor value gets too high or if another sensor value has recorded a value out of range more than twice in the last 24 hours. Similarly, one might want to notify the campus police if an overhead projector is noticed to be leaving a building. Every application could potentially monitor multiple streams of data, requesting alerts if complicated conditions are met. Thus, the scale of trigger processing required in this environment far exceeds that found in traditional DBMS applications.

Fourth, stream data is often lost, stale, or imprecise. An object being monitored may move out of range of a sensor system, thereby resulting in lost data. The most recent report on the location of the object becomes more and more inaccurate over time. Moreover, sensors that record bodily functions (such as heartbeat) are quite imprecise, and have margins of error that are significant in size.

Lastly, many monitoring applications have real-time requirements. Applications that monitor mobile sensors (e.g., military applications monitoring soldier locations) often have a low tolerance for stale data, making these applications effectively real time. The added stress on a DBMS that must serve real-time applications makes it imperative that the DBMS employ intelligent and graceful degradation strategies (i.e., *load shedding*) during periods of high load.

Monitoring applications are very difficult to implement in traditional DBMSs. First, the basic computation model is wrong: DBMSs have a HADP model while monitoring applications often require a DAHP model. In addition, to store time series information one has only two choices. First, he can encode the time series as current data in normal tables. In this case, assembling the historical time series is very expensive because the required data is spread over many records, thereby dramatically slowing performance. Alternately, he can encode time series information in BLOBs to achieve physical locality, but at the expense of making queries to individual values in the time series very difficult. The only system that we are aware of that tries to do something more intelligent with time series data is the Informix Universal Server (now owned by IBM), which implemented a time-series data type and associated methods that speed retrieval of values in a time series [1].

If a monitoring application had very large number of triggers or alerters, then current DBMSs would fail because they do not scale past a few triggers per table. The only alternative is to encode triggers in some middleware

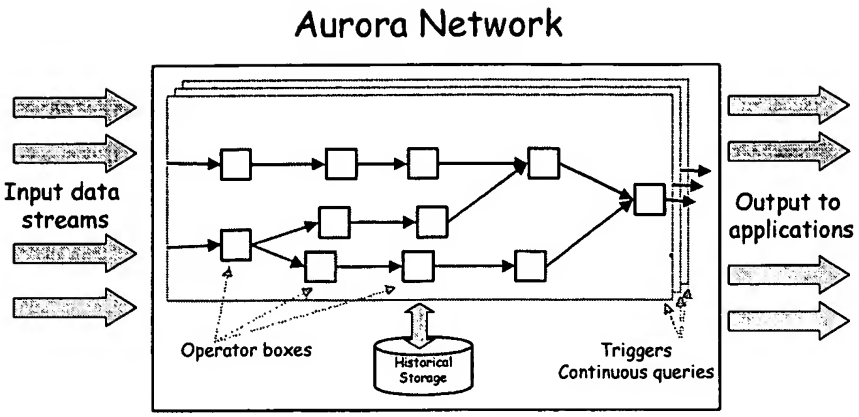


Figure 1: Aurora system model

application. Using this implementation, triggers cannot be queried because they are outside the DBMS. Moreover, performance is typically poor because middleware must poll for data values that triggers and alerters depend on.

Lastly, no DBMS that we are aware of has built-in facilities for imprecise or stale data. The same comment applies to real-time capabilities. Again, the user must build custom code into his application.

For these reasons, monitoring applications are difficult to implement using traditional DBMS technology. To do better, all the basic mechanisms in current DBMSs must be rethought. In this paper, we describe a prototype system, *Aurora*, which is designed to better support monitoring applications.

Monitoring applications are applications for which streams of information, triggers, imprecise data, and real-time requirements are prevalent. We expect that there will be a large class of such applications. For example, we expect the class of monitoring applications for physical facilities (e.g., monitoring unusual events at nuclear power plants) to grow in response to growing needs for security). In addition, as GPS-style devices are attached to a broader and broader class of objects, monitoring applications will expand in scope. Currently such monitoring is expensive and is restricted to costly items like automobiles (e.g., Lojack technology). In the future, it will be available for most objects whose position is of interest.

In Section 2, we begin by describing the basic Aurora architecture and fundamental building blocks. Section 3 continues with the design of the run-time system for Aurora. In Section 4, we show why traditional query optimization fails in our environment, and present our alternate strategies for optimizing Aurora applications. Section 5 shows how Aurora deals with the needs of real-time applications. Section 6 follows with a discussion of the notion of Aurora transactions. In Section 7, we discuss the myriad of related work that has preceded our effort. Finally, we conclude in Section 8.

2 Aurora system structure

Aurora data is assumed to come from a variety of data sources. Sometimes such data sources are computer programs that generate values at regular or irregular intervals. In other applications, values are provided directly by hardware *sensors*. We will use the term *data source* for either case. In addition, a data *stream* is the term we will use for the

collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and Aurora timestamps every incoming message to monitor the quality of service being provided.

The basic job of Aurora is to process incoming streams in the way defined by an *application administrator*. Aurora is fundamentally a data-flow system and uses the popular *boxes and arrows* paradigm found in most process flow and workflow systems. Hence, messages flow through a directed graph of processing operations (i.e., *boxes*). Ultimately, output streams are presented to *applications*, which must be programmed to deal with the asynchronous messages in an output stream. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries (which we discuss later in this section). Figure 1 illustrates this basic system structure.

Applications express their stream processing requirements by way of triggers[1]. Aurora contains built-in support for seven primitive operations for expressing its stream processing requirements. The *filter* box applies a predicate to the tuple contained in each incoming stream element, referred to as a *message*, passing only the ones that satisfy the predicate. *Merge* combines two streams of data into a single stream. *Resample* is a powerful operation that predicts additional values that are not originally contained in a stream. This operation can be used to create new values in the stream by interpolating between existing values. In addition, Aurora provides a *drop* box that removes messages from the stream, and is used primarily to shed load when Aurora is not providing reasonable service (see Section 5.3). *Join* is a windowed version of the standard relational join. Join pairs elements in separate streams whose *distance* (e.g., difference in time, relative position, etc.) is within a specified bound. This bound is considered the size of the window for the join. For example, the window might be 30 minutes if one wanted to pair all securities that had the same price within a half hour of each other. In environments where data can be stale or time imprecise, windowed operations are a necessity. The last box is called *map* and it supports the application of a function to every element of a data stream. Some functions just transform individual items in the stream to other items, while others, such as moving average, apply a function across a window of values in a stream. Hence, map has both a windowed and non-windowed version.

A precise definition for each of these operators is contained in Appendix 1. The reader can notice that both versions of map provide for user-defined functions. In this way, significant extensibility can be provided. At the current time, we do not plan for user-defined boxes, because our optimization strategies, to be presented in Section 4, rely on fairly complex knowledge about the semantics of our box types. Extensibility in this area would be tricky to support. Notice that three of the seven Aurora primitive operations (i.e., resample, join, and windowed map) process windows of data elements. To support such operations, Aurora must buffer previous values of a data stream. These windows define the historical storage requirements for stream data. Aurora is aware of the processing that must be done on this buffer database, and can automatically choose indexes, where appropriate. In general, this database is relatively small, and we expect to keep it in main memory to provide good response time. However, the reader should notice that the number of messages contained in a window might be variable. This complicates buffer allocation issues, which will be studied elsewhere.

---

[1] In this paper, we will ignore the trigger actions taken by applications once they receive data. Instead, we focus on the way that applications express the *query portion* of the trigger that specifies the ways that stream data must be processed before its delivery to the application.

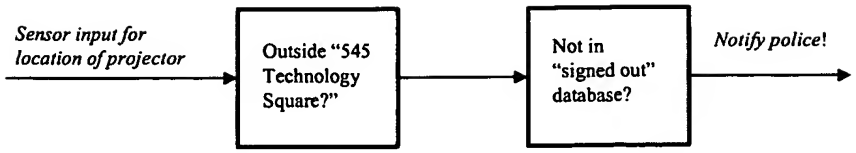


Figure 2: A simple Aurora application

The job of the application administrator is to construct a directed graph of these operations to accomplish a specific monitoring task. Figure 2 shows a simple Aurora application that tracks the location of an overhead transparency projector and notifies the police if it is in an improper location. It processes an input stream detailing the location of the projector. First, it filters the stream to only those data values that are outside the building where the projector is located (in this case "545 Technology Square"). The stream is further filtered to only contain a value if the projector is not checked out. If both conditions are true, Aurora sends a message to an external application that will notify the campus police. The processing required of Aurora is to *push* each input from a data source through the network and generate output messages in a timely fashion.

Every Aurora network is required to be free from loops. A graph with cycles has unpredictable behavior, and we wish to avoid this situation. Of course, it is possible for an application that receives a message to update a data source control that in turn modifies the future behavior of an input stream. Hence, there may be dependencies between output streams and input streams; however, these dependencies involve an intervening application program that is external to Aurora. As a result, Aurora is unaware of this dependency.

Furthermore, there are no condition boxes in an Aurora network. Instead, an application would have to include two filter boxes providing both the condition and the converse of the condition. At some inconvenience to an application, this results in a simpler system structure. Additionally, there is no explicit *split* box; instead, the application administrator can connect the output of one box to the input of several others. This implements an implicit split operation. On the other hand, there is an explicit Aurora merge operation, whereby two streams can be put together. If, additionally, one message must be delayed for the arrival of a second one, then a resample box can be inserted in the Aurora network to accomplish this effect.

One can view an Aurora network as a large collection of triggers. Each path from a sensor input to an output can be viewed as computing the *condition* part of a complex trigger. An output message is delivered to an application, which can take any appropriate action. Hence, an Aurora network is a large collection of (possibly) complex triggers. It is certainly possible for more than one of these streams to have a value at the same time. For example, if both high temperature and high humidity are true, then two of the above example streams will have values. In this case, the objective of Aurora is to present both streams to the application in a timely fashion. The decision on which stream to act on first (i.e., prioritization) rests with the application and is outside the scope of Aurora.

Furthermore, Aurora can also easily implement so-called continuous queries. Any path through the network can correspond to the processing normally involved in such a query. If output messages are directed to an application that renders them onto the screen, then a user watching the screen will see the result of a continuous query. Of course, Aurora processing of continuous queries is radically different from what is required in conventional systems (e.g., [32]). Streams from one or more data sources are pushed by Aurora to a rendering application. In contrast, a

conventional system records many user updates to a DBMS, and complex logic is required to figure out when new records qualify for the predicate of a continuous query.

Another issue is Aurora support for ad-hoc queries; i.e., ones not designed by the application administrator and hence not present in the initial Aurora network. These are accomplished by an end-user (or his agent) adding additional boxes to a running Aurora network. Such boxes are different from normal Aurora boxes because they are removed from the network when the application connected to them terminates. As such, they are useful in providing a snapshot of Aurora behavior.

If the user requests historical information in his ad-hoc query, (e.g., the duty cycle of a slide projector over the last month) then Aurora will only be able to provide an answer if the appropriate information has been retained. To this end, the application administrator can request Aurora to keep more information than would normally be required to provide semantically correct network processing. One aspect of the design-time environment is a facility to specify such larger windows. Ultimately, a window can be made infinite, and Aurora asked to "save everything."

The Aurora user interface cannot be covered in detail because of space limitations. Here, we mention only a few salient features. To facilitate designing large networks, Aurora will support a hierarchical collection of groups of boxes. A designer can begin near the top of the hierarchy where only a few *superboxes* are visible on the screen. A *zoom* capability is provided to allow him to move into specific portions of the network, by replacing a group with its constituent boxes and groups. In this way, a browsing capability is provided for the Aurora diagram. Groups also provide a convenient abstraction for alternate implementations of functionality. Often an application has a "fast and sloppy" implementation as well as a "slow and careful" one for a portion of diagram processing. One may want to switch between implementations, depending on system load. Our concept of groups will support this construct.

Boxes and groups have a tag, an argument list, a description of the functionality and ultimately a manual page. By querying these attributes a user can "teleport" to specific places in an Aurora network. Additionally, a user can place *bookmarks* in a network to allow him to return to places of interest.

These capabilities give an Aurora user a mechanism to query the Aurora diagram, thereby satisfying one of the design goals in the introduction, namely query facilities for the trigger base.

The user interface also allows monitors for arcs in the network to facilitate debugging, as well as facilities for "single stepping" through a sequence of Aurora boxes. We plan a graphical performance monitor, as well as more sophisticated query capabilities.

3 The Aurora run-time environment

The basic purpose of the execution engine is to process data flows through a potentially large workflow diagram. Although the size of the network depends on the application, our discussion with potential Aurora users indicates requirements between 10^5 and 10^7 data sources and a like number of boxes. Hence, allocating a thread for each processing box in a network will not scale to large Aurora networks. Instead, we propose an alternate approach. We plan to compile the network into a main memory data structure that can be thought of as the following tables:

- *Boxes* (box-id, box-definition, pointer-to-box-state)
- *Connections* (box-id, successor-box-id)

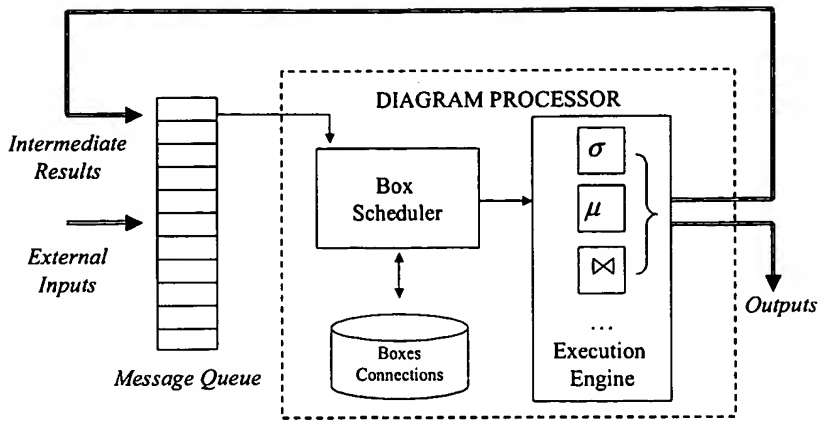


Figure 3: Aurora run-time architecture

The Boxes table contains the definition and state for each box. The state includes parameters for the box and any historical messages required for correct window processing. Whenever two boxes share overlapping state (e.g., one requires a window for elements that arrive in 10-minute intervals, while the second requires a window for elements that arrive in 8-minute intervals), the compiler will notice the shared state and allocate the larger state just once. Multiple boxes can then share the required data structure. The Connections table contains the arcs in the network.

Further, we are planning to implement the run-time architecture of Figure 3. Here, inputs from data sources are marked with an arc-id and a timestamp, and added to a (potentially large) queue. A multi-threaded diagram processor consists of a scheduler and an execution engine. The scheduler picks an element from the queue, ascertains what processing is required and passes the queue message, together with a pointer to the box-description and state to the correct box processor (which is also multi-threaded). There is one box processor for each kind of Aurora operation (e.g., join, map, etc.). The box processor performs the appropriate operation and then puts an output message on the queue. The diagram processor then ascertains the next processing step and the cycle repeats. If the message must be sent to an external application as an output, then the diagram processor performs the appropriate action. We expect to achieve high performance on large computer systems by allocating multiple threads for the diagram interpreter and each box processor. Another motivation for implementing our own scheduler (instead of using the built-in thread scheduler) is that our scheduler will be able to react to the semantic character of the network, allocating resources in Aurora-specific ways.

4 Aurora optimization

In traditional relational query optimization, one of the primary objectives is to minimize the number of iterations over large data sets. Stream-oriented operators that constitute the Aurora network on the other hand, are designed to operate in a data flow mode in which data elements are processed as they appear on the input. Although the amount of computation required by an operator to process a new element is usually quite small, we expect to have a very large number of boxes in the Aurora workflow diagram that represents all of the triggers and ad-hoc queries. Furthermore, high data rates add another dimension to the problem. In this section, we present our strategies to

construct an optimized Aurora network based on static information such as estimated cost of the boxes and selectivity statistics. Real-time optimizations, which mostly concern the data rates, are presented in Section 5.

Combining all the boxes into a massive query and then applying conventional query optimization is not a workable approach for the Aurora system; however, some conventional principles still apply. As in multiple-query optimization [28], one way to go about optimizing multiple triggers is to try to globally optimize them, which is shown to be NP-complete [29]. Instead, we focus on the following collection of alternative tactics to come up with locally optimal triggers which can then be compiled into a single network of boxes (also using common subexpression elimination techniques where possible):

1. Inserting projections
2. Combining boxes
3. Reordering boxes
4. R-tree processing
5. Trading computation for storage
6. Special techniques for ad-hoc queries

The first four tactics deal with rearrangement and efficient processing of boxes, while the fifth lowers memory consumption when main memory buffering is an issue. The final optimization addresses the addition of ad-hoc queries to an Aurora diagram. In the rest of this section, we explore the above tactics in detail.

4.1 Inserting projections

It is unlikely that the application administrator will have inserted map operators to project out all unneeded attributes. Examination of an Aurora network allows us to insert/move such map operations to the earliest possible points in the network, thereby shrinking the size of the messages that must be subsequently processed. Note that this kind of an optimization requires that the system be provided with function/operator signatures that describe the attributes that are used and produced by the operators.

4.2 Combining boxes

As a next step, Aurora diagrams will be processed to combine boxes where possible. A pair-wise examination of the operators suggests that, in general, map and filter can be combined with almost all of the operators whereas windowed or binary operators cannot.

It is desirable to combine two boxes into a single box when this leads to some cost reduction. As an example, a map operator that only projects out attributes can be combined easily with any adjacent operator, thereby saving the box execution overhead for a very cheap operator. In addition, two filtering operations can be combined into a single, more complex filter that can be more efficiently executed than the two boxes it replaces. Not only is the overhead of a second box activation avoided, but also standard relational optimization on one-table predicates can be applied in the larger box. Likewise, two map operations can be combined. Although additional operations can be combined without affecting the semantics of the outcome (such as merge and filter), there may not be a noticeable gain. In general, combining boxes at least saves the box execution overhead, and reduces the total number of boxes, leading to a simpler diagram. The next step is to change the order of box processing where advantageous and possible.

4.3 Reordering boxes

Reordering the operations in a conventional relational DBMS to an equivalent but more efficient form is a common technique in query optimization. For example, filter operations can sometimes be pushed down the query tree through joins. In Aurora, we can apply the same technique when two operations are commutative.

	filter	map	w. map	merge	resample	join	drop
filter	*yes*						
map	*maybe*	*maybe*					
w. map	*no*	*maybe*	*maybe*				
merge	*yes*	*yes*	*no*	*yes*			
resample	*maybe*	*maybe*	*no*	*no*	*maybe*		
join	*maybe*	*maybe*	*no*	*no*	*no*	*maybe*	
drop	*yes*	*yes*	*no*	*yes*	*no*	*no*	*yes*

Table 1: Operator commutativity table

We present the commutativity of each pair of primitive boxes in Aurora in Table 1. We observe that a few operators always commute, while many commute under certain circumstances and a few never commute. The operations that conditionally commute usually depend on the size of the window. As one would expect, the operators with window size one are more well behaved than the ones with larger windows are. In some cases, the format of the input can also play a role. In Appendix 2, we examine the commutativity of our operations in detail.

To decide when to interchange two commutative operators, we make use of the following performance model. Each Aurora box, b, has a *cost*, $c(b)$, defined as the expected execution time for b to process one input message and produce one output message. Additionally, each box has a *selectivity*, $s(b)$, which is the expected number of output messages per input message. For instance, filter will always have a selectivity less than or equal to one, whereas resample and join may have a selectivity greater than one.

Consider two boxes, b_i and b_j, with b_j following b_i. In this case, for each input message for b_i, we can compute the amount of processing as $c(b_i) + c(b_j) \times s(b_i)$. Reversing the operators gives a like calculation. Hence, we can compute the condition used to decide whether the boxes should be switched as:

$$\frac{1-s(b_j)}{c(b_j)} > \frac{1-s(b_i)}{c(b_i)}$$

The above condition says that if the ratio of the rate reduction caused by b_j to its cost is greater than the corresponding ratio for b_i, then b_j should be executed before b_i. It is straightforward to generalize the above calculation to deal with cases that involve fan-in or fan-out situations. Moreover, it is easy to see that we can obtain an optimal ordering by sorting all the boxes according to their corresponding ratios in decreasing order (i.e., the box with the highest rate reduction-to-cost ratio should be the one closest to data sources). We can use this result to come up with a heuristic algorithm that iteratively reorders boxes (to the extent allowed by their commutativity properties) until no more reorderings are possible.

4.4 R-tree processing

Consider the situation that a box output is split a large number of times and sent to a large collection of subsequent filter boxes. In a chemical plant, for example, there might be many different actions to take, depending on the value of an input sensor. In this case, Aurora can assemble all the filters into a single superbox, supported in the following way.

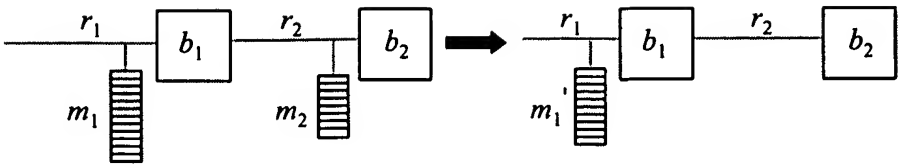


Figure 4: Trading computation for storage

All the predicates in all the boxes are examined and an R-tree [15] is implemented on all the attributes that appear in some predicate. If there is too much predicate diversity, then the *outliers* are tossed out of the superbox to be processed in the normal fashion. Hence, each predicate in the superbox is a rectangle in this R-tree. If there are k filters in the superbox, then the R-tree has k rectangles.

Whenever the superbox receives an input, it goes through the motion of inserting the corresponding point in this R-tree in order to find which rectangles cover this point. It then activates only those boxes corresponding to the correct rectangles.

This concept can be generalized somewhat. For example, if a box requires two values in a certain range within a certain time period, then the rectangle can include time as a dimension. R-tree processing must then actually insert the points into the R-tree and then activate the box only when it counts the correct number of points inside the appropriate rectangle.

4.5 Trading computation for storage

As mentioned earlier, windowed operations require Aurora to maintain a database of historical values of the recent past. This database may be small enough that it fits or nearly fits in main memory. When the size of the database is larger than the amount of available main memory, the algorithms in this section should be used. These algorithms compress the required amount of storage, at the possible expense of extra processing. The basic tactic is to move one or more data sets in this database from their current location between two operators, to an earlier position in the diagram. This will force Aurora to reprocess intervening boxes, thereby adding to the processing load. On the other hand, the total amount of storage required is potentially reduced.

Due to space considerations, we restrict our discussion to two scenarios with the greatest potential storage gain. Our approach, however, is general and can be applied to other cases as well. Let r_i represent the input message rate for box b_i. Further, let $w(m_i)$ represent the window size and $t(m_i)$ represent the width of a message in the database. The total window size for b_i is then equal to $t(b_i) \times w(b_i)$. Furthermore, let m_i' be the resulting database after storage has been moved forward in the diagram.

Our first scenario involves two boxes that are serially connected. We first demonstrate our approach using the simple example illustrated in Figure 4. Here, we observe two boxes, b_1 and b_2, that implement windowed map operations performing an aggregate function (e.g., *average*). Assume that r_1 equals r_2 (i.e., selectivity of b_1 is 1). If b_1 has a window, m_1, with the size of four, and b_2 has a window, m_2, with the size of three, then the total size of the composite windows is seven. However, m_2 can be moved forward and combined with m_1 into a single window, m_1'. This move reduces the total amount of storage required from seven messages to six. Whenever b_2 requires a message

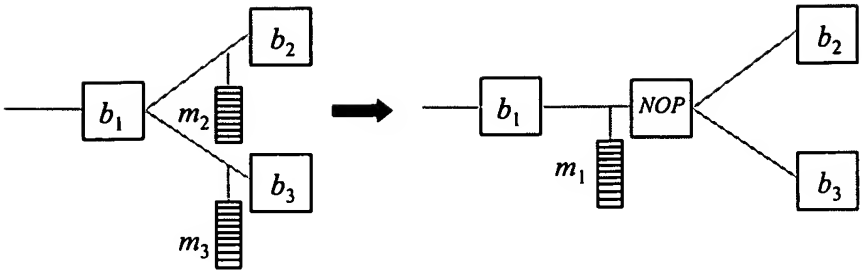


Figure 5: Fan-out case

from b_1, b_1 must *re*create the message using the messages stored in m_1'. This requires b_1 to be called two additional times. The result of combining b_2's window with b_1's is a savings of one message slot but an increase in computation of $2 \times c(b_1)$, where $c(b_1)$ represents the computation cost of b_1 as described earlier.

We can generalize the above discussion in a straightforward manner, and compute the pre- and post-move computational costs as:

$$c(b_1)+c(b_2) \text{ and } w(m_2)\times c(b_1)+c(b_2)$$

Similarly, we can also compute the pre- and post-move storage requirements as:

$$w(m_1)\times t(m_1)+w(m_2)\times t(m_2), \text{ and}$$
$$w(m_1')\times t(m_1') = (r_1 / r_2)\times(w(m_2)+w(m_1)-1)\times t(m_1)$$

Using these equations, we can easily compute the storage gained and extra computation required. If the ratio of these two values is above a certain threshold T, then we decide to move the storage.

The second case we discuss involves a fan-out situation, where the output of one operation feeds data to multiple subsequent windowed operations. In this case, which is illustrated in Figure 5, Aurora will always add a *no-op* operation between the operations and move storage in front of no-op. This will result in considerable savings from having a single shared storage rather than multiple ones at no additional computational cost. The new window size will be equal to that of the operation with the largest window size. Unlike the previous case, where there was a trade off between storage and computation, this case is *always* advantageous.

We plan to apply the above techniques to an Aurora diagram and move storage iteratively until no further moves are possible. Although this will not always produce an optimal assignment of memory to an Aurora diagram, it is an effective heuristic that is computationally feasible.

4.6 Ad-hoc query optimization

Ad-hoc queries, which usually refer to historical values, are temporarily inserted into an Aurora network after the network has been constructed, optimized, and already started running. When an ad-hoc query is inserted into a running Aurora network, there are two issues to consider. First, if the application administrator has specified historical storage of earlier messages in the network, then Aurora must generate the correct answer to the historical query posed. Logically, all relevant previously-stored upstream messages must be processed by the newly added boxes. Second, the added boxes must then continue processing live Aurora messages until they get deleted. The second issue can be dealt with by the optimizations discussed in Sections 4.1– 4.5. Hence, we now discuss how we can process the historical part of an ad-hoc query efficiently.

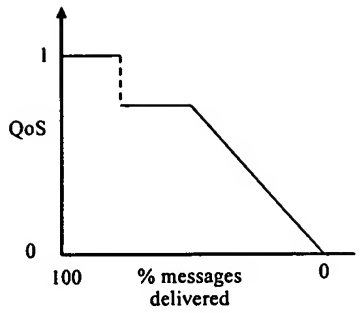


Figure 6: A message-based QoS graph

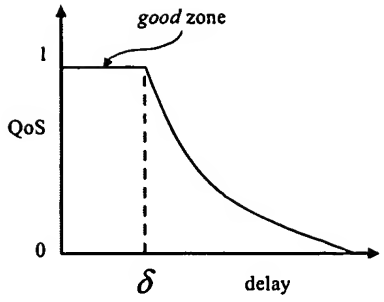


Figure 7: A delay-based QoS graph

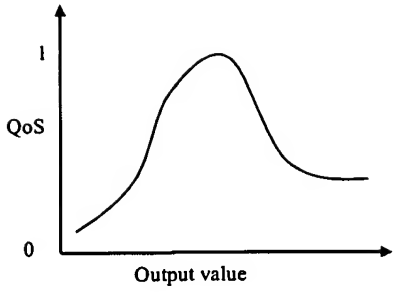


Figure 8: A value-based QoS graph

There are two possible solution tactics. First, the stored messages can be pushed through the extra network, after the optimizations of Sections 4.1– 4.5 have been applied. Since Aurora knows the cost and selectivity of each box and the number of historical messages stored, the overall cost of this tactic can be easily computed. Alternately, Aurora can combine all the boxes into a single query, optimize the query using standard relational techniques, and then run the query against the stored historical dataset(s). This will implement a *pull* strategy, whereby only valid messages are pulled from the dataset(s) and sent to the output application. Using conventional techniques, the expected cost of this pull tactic can also be computed.

The plan for Aurora is to use the cheaper of these two techniques to solve the historical portion of the query. Of course, there is also a synchronization issue because the result of the historical query must be integrated with the continuing answer to be subsequently generated.

5 Real-time operation

This section is concerned with maintaining quality of service in a real-time environment. In Section 5.1, we begin with the auxiliary data structures that are required by the algorithms presented in this section. In Section 5.2, we first present a static analysis to determine if Aurora system resources are adequate to handle the expected system load. We then turn to investigating the dynamic behavior of an Aurora system and discuss the issue of determining whether an Aurora system is overloaded due to long-duration load spikes that result from increased data source activity. In Section 5.3, we present our load shedding algorithms that intelligently drop messages from an Aurora network to ensure that the expected data arrival rates will not overload system resources in steady state and that the best possible quality of service is achieved for Aurora outputs even under unexpected loads. In Section 5.4, we present a summary of Aurora's overall approach to introspection and load shedding.

5.1 Data structures

When the input rates exceed system resource limits, load must be shed to maximize the perceived quality of service (QoS) for the outputs produced by Aurora. QoS, in general, is a multidimensional function of several attributes of an Aurora system. These include:

- *response times*— output messages should be produced in a timely fashion; as otherwise QoS will degrade as delays get longer;
- *message drops*—if messages are dropped to shed load, then the QoS of the affected outputs will deteriorate;

- *values produced*—QoS clearly depends on whether important values are being produced or not.

Asking the application administrator to specify a multidimensional QoS function seems unworkable. Instead, Aurora relies on a simpler tactic, which is much easier for humans to deal with: for each output stream, we expect the application administrator to give Aurora a two-dimensional QoS graph based on the percentage of messages delivered (as illustrated in Figure 6). In this case, the application administrator indicates that high QoS is achieved when message delivery is near 100% and that QoS degrades as messages are dropped.

Optionally, the application administrator can give Aurora either one of two additional QoS graphs for all outputs in an Aurora system. The first, illustrated in Figure 7, is a delay-based QoS graph. Here, the QoS of the output is maximum if delay is less than the threshold, δ, in the graph. Beyond δ, QoS degrades with additional message delay. The application administrator should present Aurora with the information in Figure 7 for each output, if he is capable of doing so.

Aurora also assumes that all quality of service graphs are normalized, so that we can compare QoS for different outputs quantitatively. Aurora further assumes that the value chosen for δ is *feasible*, i.e., that a properly sized Aurora network will operate with all outputs in the *good* zone to the left of δ. This will require the delay introduced by the total computational cost along the longest path from a data source to this output not to exceed δ. If the application administrator does not present Aurora with feasible QoS graphs, then the algorithms in the subsequent sections will not produce good results.

The second optional QoS graph for outputs is shown in Figure 8. The possible values produced as outputs appear on the horizontal axis, and the QoS graph indicates the importance of each one. This value-based QoS graph captures the fact that some outputs are more important than others. For example, in a plant monitoring application, outputs near a critical region are much more important than ones well away from it. Again, if the application administrator has value-based QoS information, then Aurora will use it to shed load more intelligently than would occur otherwise.

The last item of information required from the application administrator is H, the *headroom* for the application, defined as the percentage of the computing resources that can be used in steady state. The remainder are reserved for the expected ad-hoc queries, which are added dynamically.

In summary, Aurora requires the application administrator to specify the headroom as well as message-based QoS graphs for each output. In addition, the administrator can optionally give Aurora delay-based or value-based QoS graphs for all outputs. In the rest of this section, we discuss how to use these QoS graphs to ensure adequate steady-state performance and handle spikey input data rates by intelligently shedding load.

5.2 Introspection

Aurora employs static and run-time introspection techniques to predict or detect potential overload situations.

5.2.1 Static analysis

The first task for Aurora is to determine if the hardware running the Aurora network is sized correctly. If insufficient computational resources are present to handle the steady state requirements of an Aurora network, then queue lengths will increase without bound and response times will become arbitrarily large.

As described before, each box, b, in an Aurora network has an expected message processing cost, $c(b)$, and a selectivity, $s(b)$. If we also know the expected rate of message production $r(d)$ from each data source d, then we can use the following static analysis to ascertain if Aurora is sized correctly.

From each data source, we begin by examining the immediate downstream boxes: if box b_i is directly downstream from data source d_i, then, for the system to be stable, the throughput of b_i should be at least as large as the input data rate; i.e.,

$$\frac{1}{c(b_i)} \geq r(d_i)$$

Moreover, we can calculate the output data rate from b_i as:

$$min(\frac{1}{c(b_i)}, r(d_i)) \times s(b_i)$$

Proceeding iteratively, we can compute the output data rate and computational requirements for each box in an Aurora network. We can then calculate the minimum aggregate computational resources required per unit time, *min_cap*, for stable steady-state operation. Clearly, the Aurora system with a capacity C cannot handle the expected steady state load if C is smaller than *min_cap*. Furthermore, the response times will assuredly suffer under the expected load of ad-hoc queries if

$$C \times H < min\_cap$$

Clearly, this is an undesirable situation and can be corrected by redesigning applications to change their resource requirements, by supplying more resources to increase system capacity, or by load shedding (see Section 5.3).

5.2.2 Dynamic analysis

Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable and long-duration spikes in input data rates may deteriorate overall system performance to a level that renders the system useless. We now describe two run-time techniques to detect such cases.

Our first technique for detecting an overloaded system relies on the use of delay-based QoS information, if available. Aurora timestamps all messages from data sources as they arrive. Furthermore, all Aurora operators preserve the message timestamps as they produce output messages (if an operator has multiple input messages, then the earlier timestamp is preserved). When Aurora delivers an output message to an application, it checks the corresponding delay-based QoS graph (see Figure 7) for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone). Of course, a single errant output is not cause for alarm, so Aurora watches to see if the number of errant outputs exceeds a threshold to decide whether the system is behaving abnormally and corrective action needs to be taken.

If delay-based QoS information is not available to guide Aurora in detecting abnormal operation, then Aurora employs a somewhat cruder technique. Specifically, Aurora watches its internal dispatching queue for evidence of sustained growth of one or more of the logical queues (in front of some box(es)). If a buildup is observed for a period of time exceeding a threshold, then Aurora takes corrective action. Of course, longer queue lengths do not necessarily mean that the delivered QoS is bad or unacceptable. Unless delay-based QoS information is available, however, this is the most reliable piece of information on which to base load shedding decisions.

5.3 Load shedding

When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora message processing via load shedding. The naïve approach to load shedding involves dropping messages at randomly picked points in the network and in an entirely uncontrolled manner. Such an approach has two potential problems: (1) overall system utility, which is defined by the QoS graphs, might be degraded much more than necessary (or, depending on the type of QoS graph used, might be improved much less than possible); and (2) application semantics might be arbitrarily affected. In order to alleviate these problems, Aurora relies on the QoS graphs to guide its load shedding process. We now describe two load shedding schemes that differ in the way they exploit QoS information.

5.3.1 Load shedding by dropping messages

Our first load shedding approach addresses the former problem mentioned above: it attempts to minimize the degradation (or maximize the improvement) in the overall system QoS (i.e., the QoS values aggregated over all the outputs). This is accomplished by dropping messages on network branches that terminate in *more tolerant* outputs.

If load shedding is triggered as a result of static analysis, then we cannot expect to use delay-based or value-based QoS information to guide load shedding (without assuming the availability of a priori knowledge of the message delays or frequency distribution of values). Therefore, for the static analysis case, we can only use message-based QoS graphs. If load shedding is triggered as a result of dynamic analysis, however, we can use delay-based QoS graphs if they are available.

We employ a greedy algorithm to perform load shedding as follows (without loss of generality, we initially assume that message-based QoS graphs are used to drive the load shedding process). We first identify the output with the *smallest* negative slope for the corresponding QoS graph. We move horizontally along this curve until there is another output whose message-based QoS curve has a smaller negative slope at that point. This horizontal difference gives us an indication of the *output* messages to drop (i.e., the selectivity of the drop box to be inserted) that would result in the minimum decrease in the overall QoS. We then move the corresponding drop box as far upstream as possible until we find a box that affects other outputs (i.e., a *split point*), and place the drop box at this point in the network. Meanwhile, we can calculate the amount of recovered resources or the new delay values that result from this exercise. If the system resources are still not sufficient or response time goals are not met, then we identify another output and repeat the process.

If we used a delay-based QoS graph instead, we would then be interested in identifying the output with the *largest* negative slope (i.e., the output that would yield the maximum increase in overall QoS). The rest of the algorithm would proceed similarly.

Presumably, the application is coded so that it can tolerate missing messages from a data source caused by communication failures or other problems. Hence, this mechanism for shedding load just artificially introduces additional missing messages. Although the semantics of the application are now different, the harm should not be too damaging.

5.3.2 Semantic load shedding by filtering messages

The load shedding scheme described above effectively reduces the amount of Aurora processing by dropping *randomly selected* messages at strategic points in the network. While this approach attempts to minimize the loss in overall system utility, it fails to control the impact of the dropped messages on application semantics. Semantic load shedding addresses this limitation by using value-based QoS information, if available. Recall that this QoS graph provides information about the relative importance of various values for a given output. Specifically, semantic load shedding drops messages in a more controlled way; i.e., it drops less important messages, rather than random ones, using filter boxes (instead of drop boxes).

If value-based QoS information is available, then Aurora can watch each output and build up a histogram containing the frequency with which value ranges have been observed. In addition, Aurora can calculate the expected utility (QoS) of a range of outputs by multiplying the QoS values with the corresponding frequency values for every interval and then summing these values. To shed load, Aurora identifies the output with the *lowest utility interval*; converts this interval to a filter predicate; and then, as before, attempts to propagate the corresponding filter box as far upstream as possible to a split point. This strategy, which we refer to as *backward interval propagation*, admittedly has limited scope because it requires the application of the inverse function for each operator passed upstream (Aurora boxes do not necessarily have inverses). An alternative strategy, *forward interval propagation, estimates* a proper filter predicate and propagates it in *downstream* direction to see what results at the output. By trial-and-error, Aurora can converge on a desired filter predicate. Note that a combination of these two strategies can also be utilized. First, Aurora can apply backward propagation until a box, say *B*, whose operator's inverse is difficult to compute. Aurora can then apply forward propagation between the insertion location of the filter box and *B*. This algorithm can be applied iteratively until sufficient load has been shed.

5.4 Summary

Aurora's approach to introspection and load shedding can be outlined as follows. We use static analysis to correctly size an Aurora network. If the system lacks sufficient resources, we use message-based QoS graphs to place drop boxes into an Aurora network in advance of operation. If all outputs have value-based QoS graphs, we can also transform the drop boxes to appropriate filter boxes once the network is operational. This will ensure adequate steady-state operation.

Dynamically, there is also a need to shed load to deal with unpredictably heavy loads. We can use delay-based QoS for each output to identify when load must be shed. If this information is not available, then we must use cruder queue-length metrics. If all outputs have value-based QoS graphs, then we can insert filter nodes into an Aurora network; otherwise, we insert drop nodes using delay-based QoS information.

Of course, Aurora will contain heuristics to delay shedding load until bad behavior has been observed for a while. Likewise, good behavior must also persist before load shedding is reversed. Note that we can reverse load shedding simply by removing the drop (or filter) boxes in the reverse order they were inserted into network.

In summary, we can perform static and run-time analysis to detect (potential) overload situations and perform load shedding to decrease the number of messages Aurora needs to process in those cases. Depending on the

availability of certain QoS information, Aurora can perform either standard load shedding using drop boxes or more controlled semantic load shedding using filter boxes.

6 Transaction processing

It is clear that Aurora can shed load if required to meet responsiveness goals. A crash is really nothing but a massive shedding of load. Hence, recovering from crashes does not necessarily require standard write ahead logging and redo/undo techniques. Instead, Aurora will simply resume processing with as much preserved data as possible. Hence, Aurora develops *partial amnesia* upon crash occurences. If the disk is not intact, then Aurora's amnesia may become total.

Similarly, Aurora needs a semaphore for the top of the message queue. In addition, when Aurora rearranges the elements of the queue, it must implement some sort of locking scheme. The same comment applies when a box updates any state that it shares with one or more other boxes. However, there is no concept of transaction abort or deadlock. As such, Aurora transaction management can be very simple, and will not entail a lot of complexity.

7 Related work

In this section, we summarize previous research related to Aurora. In particular, we discuss efforts on continuous queries, triggers, adaptive query processing, stream processing, materialized views, and approximate query answering.

A special case of Aurora processing is as a continuous query system. Many continuous query systems have focused their interests in performance on the topic of indexing the queries (e.g., [3]). In contrast, Aurora is a much more general effort. However, we plan to examine previous query indexing systems for applicability in the Aurora environment. A system like Niagara [7] is concerned with combining multiple data sources in a wide area setting while we are initially focusing on the construction of a stream server that can process very large numbers of streams. Niagara also focuses on Internet-style data such as XML and other semi-structured sources while our emphasis is on data sources that produce time-series.

As in Aurora, active databases [26, 27] are concerned with monitoring conditions. These conditions can be a result of any arbitrary update on the stored database state. In our setting, updates are append-only, thus requiring different processing strategies for detecting monitored conditions. Triggers evaluate conditions that are either true or false. Our framework is general enough to support generalized queries over stream or the conversion of these queries into monitored conditions. There has also been extensive work on making active databases highly scalable (e.g., [16]). Similar to continuous query research, these efforts have focused on query indexing, while Aurora is constructing a more general system.

Adaptive query processing techniques (e.g., [4, 17, 21, 33]) address efficient query execution in unpredictable and dynamic environments (e.g., Internet data sources) by revising the query execution plan as the characteristics of incoming data changes. The adaptation approach we described in this paper, load shedding, is radically different from previous adaptive techniques in that it involves dropping heuristically selected data from an incoming data stream, albeit in a controlled manner. It is possible that adaptive query processing is also useful for efficient processing of stream data and we plan to investigate the utility of this complementary technique in the future.

Directly related work on stream data query processing architectures shares many of the goals and target application domains with Aurora. The Streams project [5] attempts to provide complete DBMS functionality along with support for continuous queries over streaming data. Aurora's main emphasis is on efficiently supporting sophisticated continuous queries over a large number of potentially very fast data streams, thus sacrificing some traditional database functionality (e.g., transaction management, storage management, etc.) for extreme scalability. The Fjords architecture [24] addresses continuous query execution strategies that combine push-based sensor data with data from traditional sources that produce data using a pull-based, blocking interface.

Tools for data mining of stream-based data have received considerable attention lately. Some tools only allow one pass over stored stream data, for example [19]. Their main motivation is to be able to handle standard graph problems in one pass. Others [9, 20, 35] are interested in addressing data mining problems and, thus, require multiple passes over stream data. In contrast to efforts that are performing historical analysis of streams, Aurora is focused on monitoring streams in real time.

Work in sequence databases [30, 31] defined sequence definition and manipulation languages over discrete data sequences (typically by assigning each incoming tuple with a specific point in time). The Chronicle data model [22] defined a restricted view definition and manipulation language over append-only sequences. Aurora's algebra extends relevant aspects of previous proposals by proposing a binary windowed operator (i.e., join), which are indispensable for continuous query execution over data streams.

Our work is also relevant to materialized views [14], which are essentially stored continuous queries that are re-executed (or incrementally updated) as their base data are modified. However, Aurora's notion of continuous queries differs from materialized views primarily in that Aurora updates are append-only; query results are streamed (rather than stored); and high stream data rates may require load shedding or other approximate query processing techniques that trade off efficiency for result accuracy.

Systems have been proposed that address the processing of large scientific data sets that represent streams. These systems focus on processing stored data as opposed to real-time data and provide a stream-based programming model that is at a much lower level of abstraction than our set of operations. For example, DataCutter [6] provides a stream abstraction that supports reading a section of the stream and not higher-level operators like Select or Join. However, DataCutter has investigated the use of multidimensional indexing techniques to support range queries over very large stored data sets. Hence, it has similarities to our proposed R-tree optimizations.

Previous work investigated techniques that trade off the efficiency of query execution with the accuracy of query results. The basic approach is to compute approximate answers (with corresponding confidence bounds) to ad-hoc queries by scanning subsets of base data at query time (e.g., [18]) or by avoiding accesses to base data through the use of pre-computed data summaries (e.g., [2]). Clearly, these work do not directly apply to continuous queries over data streams and, thus, to Aurora. Recent work addressed approximate answers to queries over data streams. [12] uses histogram-based techniques to compute approximate answers *correlated aggregation* queries. [13] describes wavelet-based techniques to compute data summaries, which are then used to provide approximate answers to ad-hoc queries. We believe that these and similar techniques can profitably be used in conjunction with Aurora's

dynamic load shedding approach (which also produces approximate query results), and plan to further explore them in future.

Our work is likely to benefit from and contribute to the considerable research on temporal databases [25], since Aurora has a fundamental temporal aspect to it. Likewise, we can benefit from the literature on real-time databases [23, 25], and main-memory databases [11]. In a real-time database system, transactions are assigned timing constraints and the system attempts to ensure a degree of confidence in meeting these timing requirements. Hence, this work deals with a form of load shedding. However, the Aurora notion of QoS specification extends the soft and hard deadlines employed in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual ad hoc transactions, whereas Aurora associates its QoS curves with the output from stream processing and, thus, has to support continuous timing requirements.

Main-memory databases are often used in applications with high performance requirements. They assume that most, if not all, operational data is kept in main memory at all times. However, they assume a HADP model, whereas Aurora proposes a DAHP model that builds streams as fundamental Aurora objects.

Finally, many recent papers [5, 8, 10, 34] have addressed the data management needs of specific monitoring or tracking applications. We take inspiration from these works as we try to design a general-purpose system that could be used to efficiently support all these tasks.

8 Conclusions

Monitoring applications are those where streams of information, triggers, real-time requirements, and imprecise data are prevalent. Traditional DBMSs are based on the HADP model, and thus cannot provide adequate support for such applications. In this paper, we have described the architecture of Aurora, a DAHP system, oriented towards monitoring applications. We have presented the basic system architecture, along with the primitive building blocks for workflow processing. We followed with the structure for optimizing a large Aurora network that entails combining and reordering boxes, combining collections of filters into *superboxes* for R-tree processing, and trading computation for storage where necessary. Our approach to shedding load entails maintaining satisfactory response time for high priority outputs at the expense of more tolerant ones. This work was then generalized to account for the value of the output that is being produced—processing on low-value outputs are deferred or discarded. The paper also presents a sketch of transaction processing issues.

As we emphasized in the introduction, a large and important subset of monitoring applications, in particular those that involve sensor-generated data streams, commonly needs to process missing and imprecise data values. We are currently extending the Aurora model to support monitoring applications in the presence of such values.

The load shedding approach that we described in this paper is based on intelligently shedding data. As another promising research direction, we plan to investigate a complementary *cycle shedding* approach, where the idea is to replace *expensive* operator boxes with cheaper counterparts.

The implementation of Aurora is already underway, and we hope to have a running prototype available by the middle of next year.

References

[1] Informix White Paper. Time Series: The Next Step for Telecommunications Data Management. `http://www-4.ibm.com/software/data/informix/pubs/whitepapers/nextstep_wp.pdf`.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[3] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109-120, 2001.

[6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. In *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, 2000.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[8] B. Coifman. Identifying the Onset of Congestion Rapidly with Existing Traffic Detectors. *IEEE Transactions on Intelligent Transportation Systems (ITSC). submitted for publication.*, 2001.

[9] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, 2000.

[10] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Transactions on Networking (TON)*, 9(3):280-292, 2001.

[11] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):509-516, 1992.

[12] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, 2001.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.

[14] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[15] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, Boston, MA, 1984.

[16] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, Sydney, Austrialia, 1999.

[17] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7-18, 2000.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[19] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. Compaq Systems Research Center, Palo Alto, California Technical Report TR-1998-011, May 1998.

[20] C. Hidber. Online Association Rule Mining. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[21] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[22] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[23] B. Kao and H. Garcia-Molina, "An Overview of Realtime Database Systems," in *Real Time Computing*, W. A. Halang and A. D. Stoyenko, Eds.: Springer-Verlag, 1994.

[24] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002.

[25] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):513-532, 1995.

[26] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63-103, 1999.

[27] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991.

[28] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23-52, 1988.

[29] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):262-266, 1990.

[30] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[31] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.

[32] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 1992.

[33] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[34] O. Wolfson, S. Chamberlain, P. Sistla, B. Xu, and J. Zhou. DOMINO: Databases fOr MovINg Objects tracking. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[35] B.-K. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online Data Mining for Co-Evolving Time Sequences. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, San Diego, CA, 2000.

Appendix 1

In this appendix, we give a formal definition of streams and the Aurora operations on streams.

A stream is a potentially infinite ordered set of tuples (elements). Elements in a stream can be identified by their index values (such as timestamps or integer positions) that specify where the element lies in the ordering. Generally, the index is some sort of encoding of time, but other indexes are certainly possible.

More formally, a *stream*, S, is a set of (index, tuple) pairs:

$$S = \{(i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n), \ldots\}$$

such that all indices in S, belong to some *index type* (see below), and all tuples in S adhere to the same schema.

The indexes used in a given stream must belong to a totally ordered type (i.e., a type for which one of $<$, $>$, or $=$ holds between any two elements). Such types are *Index Types*. Every index type must be associated with a *unit of measure*, which defines minimal increments in index values. For example, if the index type consists of timestamps (as in stock data), then the associated unit measure might be *15 min* or *1 hour* depending on the maximum arrival frequency of stock data. If an index type consists of positive integers (reflecting the relative positions of elements in the stream), then the associated unit measure would be 1. Units of measure are required in order to express distances between elements, and to define a window size within a query. Note that units of measure do not imply that streams must be periodic — they simply define the maximum frequency with which stream elements appear.

Our assumption is that a given stream source (e.g., sensors, broadcasting software, etc.) determines the index type and associated unit of measure for the elements emanating from it. This would require the stream source to have access to a clock (if the index is timestamp-based) or to maintain a counter (if the index is position-based) of the elements it generates. In addition, for streams of the same index type to be properly aligned, indices of a stream must start from a multiple of its unit measure.

An index type must provide the following operations:

- *Ordering relations* ($<$, $>$, $\leq$, $\geq$): Because the index type is a total order, it must be the case for any elements of the index type, i_1 and i_2, that either $i_1 < i_2$, $i_2 < i_1$, or $i_1 = i_2$.

- *Position Arithmetic* (+, -, %): For any index value, i, $i + k$ identifies the index element that is k units away from i. For example, if i is a timestamp-based index type with unit 1 hour, then 15:00 + 3 = 18:00. '-' is defined similarly. '%' performs modular arithmetic on index values. That is, for any index value, i and positive integer, k, $i \% k = r$ such that $0 \leq r \leq k\text{-}1$ and for some m, $i = mk + r$. Note that position multiplication is defined in the standard way in terms of position addition.
- *Distance* (Δ): For any two index values, i_1 and i_2, $\Delta(i_1, i_2)$ is the distance (in units) between them. Note that because units represent the smallest observed increment in index values, the result of an expression involving Δ will always be an integer.

Aurora defines seven primitive stream operations: filter (σ), map (μ), windowed map (M), merge (+), join ($\triangleright\triangleleft$), resample ($\rho$) and drop ($\delta$). In describing the semantics of these operators, we adopt the following shorthand notation:

Given a stream, $S = \{(i_1, t_1), (i_2, t_2), \ldots, (i_n, i_n), \ldots\}$:

1. $S[i_k] = \{(i_k, t_k) \mid (i_k, t_k) \in S\}$
2. $S[i_m \ldots i_n] = \{(i_k, t_k) \mid (i_k, t_k) \in S, i_m \leq i_k \leq i_n\}$
3. $index(i_k, t_k) = i_k$
4. $value(i_k, t_k) = t_k$

Using this notation, we define the operations as follows:

1. *Selection* (σ): Given a predicate, p, on (index, tuple) pairs, $\sigma_p(S)$ returns the set of pairs in S that satisfy p. More formally:

   $$\sigma_p(S) = \{x \mid x \in S, p(x)\}$$

2. *Map* (μ): Given a function f, that takes an (index, tuple) pair and returns another (index, tuple) pair, $\mu_f(S)$ returns the stream resulting from applying f to every element in S. More formally:

   $$\mu_f(S) = \{f(x) \mid x \in S\}$$

3. *Windowed-Map* (M): Given a function f, that takes a set of (index, tuple) pairs (the *window*) and returns another (index, tuple) pair, $M_{f,w}(S)$ returns the stream resulting from applying f to every set of elements in S whose index values differ by at most w units. More formally:

   $$M_{f,w}(S) = \{f(S') \mid \exists d(S' = S[d..d + w - 1])\}$$

4. *Merge* (+): Merge performs the union of tuples from separate but compatible streams (i.e., those that have the same index types, units of measure, and tuple schemas). More formally, for streams S_1 and S_2:

   $$S_1 + S_2 = S_1 \cup S_2$$

5. *Resample* (ρ): The resample operator predicts the values in one stream for all index values represented in another. The prediction is made based on a prediction function, f, and a window size, w, that indicates how much history from S_1 must be retained for f to be used for prediction. More formally, for any pair of streams, S_1 and S_2 with compatible index types, window size, w, and function, f, that takes any substream of S_1 and returns an (index, tuple) pair:

$$\rho_{w,f}(S_1, S_2) = \{(i, f(S')) \mid S_2[i] \neq \emptyset, S' = \{(i', x) \in S_1 \mid \Delta(i, i') \leq w\}\}$$

6. *Join* ($\triangleright\triangleleft$): The join operator joins pairs of streams with compatible indexes (i.e., the same types and unit measures). Elements from different streams are *joined* provided that the distance between them (as determined by their indexes) is within a provided window size. More formally, for any streams S_1 and S_2; predicate, p, on pairs of stream elements; function, f, that takes two (index, tuple) pairs and returns another (index, tuple) pair, and window size, w:

$$S1 \triangleright\triangleleft_{p,f,w} S2 = \{f(x, y) \mid \exists i, j(S_1[i] \subseteq S_1, S_2[j] \subseteq S_2, \Delta(i, j) \leq w, x \in S_1[i], y \in S_2[j]), p(x, y)\}$$

7. *Drop* (δ): The drop operator filters elements of streams according to their index values. Specifically, given a *period* of k units, δ_k would block all elements for every k^{th} index value. More formally, given a positive integer, k:

$$\delta_k(S) = \{(i, t) \mid (i, t) \in S, i\%k \neq 0\}$$

Appendix 2

	filter	map	w. map	merge	resample	join	drop
filter	*yes*						
map	1	4					
w. map	*no*	5	8				
merge	*yes*	*yes*	*no*	*yes*			
resample	2	6	*no*	*no*	9		
join	3	7	*no*	*no*	*no*	10	
drop	*yes*	*yes*	*no*	*yes*	*no*	*no*	*yes*

Table 2: Detailed operator commutativity table

This appendix treats conditional commutativity in more detail. In Table 2, we present the commutativity information presented in Table 1 with all the conditional cases numbered.What follows are the precise conditions for commutativity for each of the ten cases.

1. $\sigma_p(\mu_f(S)) \equiv \mu_f(\sigma_p(S))$, if attributes referred to in p are unchanged by f.
2. $\sigma_p(\rho(f, S_1, S_2)) \equiv \rho(f, \sigma_p(S_1), S_2)$, if
   i. p is always *true* (i.e., *selectivity*$(p) = 1$) on both S_1 and $\rho(f, S_1, S_2)$, or
   ii. $\mu_{index}(S_2) \subseteq \mu_{index}(\sigma_p(S_1))$.
3. $\sigma_p(\triangleright\triangleleft_{q,concat,w}(S_1, S_2)) \equiv \triangleright\triangleleft_{p \wedge q,concat,w}(S_1, S_2)$, where

$$\triangleright\triangleleft_{p\wedge q,concat,w}(S_1,S_2) \equiv \begin{cases} \triangleright\triangleleft_{q,concat,w}(\sigma_p(S_1),S_2), & \text{if } p \text{ is defined on } S_1 \text{ and } p \text{ is not defined on } S_2 \\ \triangleright\triangleleft_{q,concat,w}(S_1,\sigma_p(S_2)), & \text{if } p \text{ is not defined on } S_1 \text{ and } p \text{ is defined on } S_2 \\ \triangleright\triangleleft_{q,concat,w}(\sigma_p(S_1),\sigma_p(S_2)), & \text{if } p \text{ is defined on both } S_1 \text{ and } S_2 \\ \triangleright\triangleleft_{q,concat,w}(\sigma_{p_1}(S_1),\sigma_{p_2}(S_2)), & \text{if } p \text{ is not defined on either } S_1 \text{ or } S_2, \text{ but } p = p_1 \wedge p_2, \\ & \text{where } p_1 \text{ is defined on } S_1 \text{ and } p_2 \text{ is defined on } S_2 \\ \triangleright\triangleleft_{q,concat,w}(S_1,S_2), & \text{otherwise.} \end{cases}$$

In the above formulation, *concat*, which stands for concatenation of two tuples, is given as the default combining function of the join operator. In fact, any function that keeps the attributes of the joined tuples that are also referred by predicate p could be used instead.

4. $\mu_f(\mu_g(S)) \equiv \mu_g(\mu_f(S))$, if $fog = gof$.

5. $\mu_f(\mu_{g,w}(S)) \equiv \mu_{g,w}(\mu_f(S))$, if for any index i,

   $f(g(S[i..i+w-1])) = g(f(S[i]), f(S[i+1]), ..., f(S[i+w-1]))$.

   In other words, f should distribute over g. Note that this formulation assumes that indices act as keys.

6. $\mu_g(\rho(f,S_1,S_2)) \equiv \rho(f,\mu_g(S_1),S_2)$, if

   i. f introduces no new values in S_1 (e.g., f is a step-wise constant function), or

   ii. g is the identity function, or

   iii. $\mu_{index}(S_2) \subseteq \mu_{index}(S_1)$.

7. $\mu_f(\triangleright\triangleleft_{p,concat,w}(S_1,S_2)) \equiv \triangleright\triangleleft_{p,concat\ o\ f,w}(S_1,S_2)$, where

$$\triangleright\triangleleft_{p,concatof,w}(S_1,S_2) \equiv \begin{cases} \triangleright\triangleleft_{p,concat,w}(\mu_f(S_1),S_2), & \text{if } f \text{ is defined on } S_1 \text{ and } f \text{ is not defined on } S_2 \\ \triangleright\triangleleft_{p,concat,w}(S_1,\mu_f(S_2)), & \text{if } f \text{ is not defined on } S_1 \text{ and } f \text{ is defined on } S_2 \\ \triangleright\triangleleft_{p,concat,w}(\mu_f(S_1),\mu_f(S_2)), & \text{if } f \text{ is defined on both } S_1 \text{ and } S_2 \\ \triangleright\triangleleft_{p,concat,w}(\mu_{f_1}(S_1),\mu_{f_2}(S_2)), & \text{if } f \text{ is not defined on either } S_1 \text{ or } S_2, \text{ but } f = f_1\ of_2, \\ & \text{where } f_1 \text{ is defined on } S_1 \text{ and } f_2 \text{ is defined on } S_2 \\ \triangleright\triangleleft_{p,concat,w}(S_1,S_2), & \text{otherwise} \end{cases}$$

   and attributes referred to in p are unchanged by f.

8. $\mu_{f,w}(\mu_{g,w}(S)) \equiv \mu_{g,w}(\mu_{f,w}(S))$, if for any i

   $f(g(S[i..i+u-1]), g(S[i+1..i+u]), .., g(S[i+w-1..i+u+w-2])) =$

   $$g(f(S[i..i+u-1]), f(S[i+1..i+u]), .., f(S[i+w-1..i+u+w-2]))$$

9. $\rho(f,\rho(g,S_1,S_3),S_2) \equiv \rho(g,\rho(f,S_1,S_3),S_2)$, if $f = g$.

10. $\triangleright\triangleleft_{p,concat,w}(\triangleright\triangleleft_{q,concat,u}(S_1,S_2),S_3) \equiv \triangleright\triangleleft_{q,concat,u}(S_1,\triangleright\triangleleft_{p,concat,w}(S_2,S_3))$,

    if p is defined on attributes of S_2 and S_3, and q is defined on attributes of S_1 and S_2.

Continuous Queries over Data Streams*

Shivnath Babu and Jennifer Widom
Stanford University
{shivnath,widom}@cs.stanford.edu
http://www-db.stanford.edu/stream

Abstract

In many recent applications, data may take the form of continuous *data streams*, rather than finite stored data sets. Several aspects of data management need to be reconsidered in the presence of data streams, offering a new research direction for the database community. In this paper we focus primarily on the problem of query processing, specifically on how to define and evaluate *continuous queries* over data streams. We address semantic issues as well as efficiency concerns. Our main contributions are threefold. First, we specify a general and flexible architecture for query processing in the presence of data streams. Second, we use our basic architecture as a tool to clarify alternative semantics and processing techniques for continuous queries. The architecture also captures most previous work on continuous queries and data streams, as well as related concepts such as triggers and materialized views. Finally, we map out research topics in the area of query processing over data streams, showing where previous work is relevant and describing problems yet to be addressed.

1 Introduction

Traditional database management systems (DBMSs) expect all data to be managed within some form of persistent *data sets*. For many recent applications, the concept of a continuous *data stream* is more appropriate than a data set. By nature, a stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times.

Several applications naturally generate data streams as opposed to data sets: financial tickers, performance measurements in network monitoring and traffic management, log records or click-streams in web tracking and personalization, manufacturing processes, data feeds from sensor applications, call detail records in telecommunications, and others. Because today's database systems are ill-equipped to perform any kind of special storage management or query processing for data streams, heavily stream-oriented applications tend to use a DBMS largely as an offline storage system, or not at all. Like other relatively recent new demands on data management (e.g., triggers, objects), it would be beneficial to provide stream-oriented processing as an integral part of a DBMS. Several aspects of data management need to be reconsidered in the presence of data streams. The *STREAM* (STanford stREam datA Management) project at Stanford is addressing the new demands imposed by data streams on data management and processing techniques.

In this paper we focus on defining a solid framework for query processing in the presence of continuous data streams. We consider in particular *continuous queries* [TGNO92], which are queries that are issued once and then logically run continuously over the database (in contrast to traditional *one-time* queries which are run once to completion over the current data sets). In network traffic management, for example, continuous queries may be used to monitor network behavior online in order to detect anomalies (e.g., link congestion) and their cause (e.g., hardware failure, denial-of-service attack). Continuous queries may also be used to support load balancing or other network performance adjustments [DG00]. In financial applications, continuous queries may be used to monitor trends and detect fleeting opportunities [Tra]. Both of these applications are characterized by a need for continuous queries that go well beyond simple element-at-a-time processing, by rapid data streams, and by a need for timely online answers.

The organization of the rest of the paper is as follows:

- In Section 2 we provide a broad survey of previous work relevant to data stream processing and continuous queries. Although there has been only a handful of papers addressing the topic directly, a number of papers in related areas contain useful techniques and results.
- In Section 3 we introduce a concrete example to motivate our discussion of continuous queries over data streams.
- In Section 4 we define a general and flexible architecture for query processing in the presence of data

*This work was supported by the National Science Foundation under grant IIS-9811947, by NASA Ames under grant NCC2-5278, and by a Microsoft graduate fellowship.

streams. Also in Section 4 we use our basic architecture to specify alternative semantics for continuous queries, and to classify previous related work. We also use the architecture to clarify how continuous queries over data streams relate to triggers and materialized views.

- In Section 5 we map out, in some detail, a number of open research topics that must be addressed in order to realize flexible and efficient processing of continuous queries over data streams.

- Sections 6 and 7 discuss our vision of and plans for a general-purpose *Data Stream Management System* (*DSMS*).

2 Related Work

In this section we provide a general discussion of past work that relates in some way to continuous queries and/or data streams. A more technical analysis of some of the work will be provided in Section 4.3, after we present our basic architecture.

Continuous queries were an important component of the *Tapestry* system [TGNO92], which performed content-based filtering over an append-only database of email and bulletin board messages. The system supported continuous queries expressed using a quite restricted subset of SQL, in order to make guarantees about efficient (incremental) evaluation and append-only query results. The notion of continuous queries for a much wider spectrum of environments is formalized in [Bar99]. The *XFilter* content-based filtering system [AF00] performs efficient filtering of XML documents based on user profiles. The profiles are expressed as continuous queries in the *XPath* language [XPA99]. *Xyleme* [NACP01] is a similar content-based filtering system that enables very high throughput with a restricted query language. The *Tribeca* stream database manager [Sul96] provides restricted querying capability over network packet streams. We will revisit much of this work in Section 4.3.

The *Chronicle data model* [JMS95] introduced append-only ordered sequences of tuples (*chronicles*), a form of data stream. They defined a restricted view definition language and algebra that operates over chronicles together with traditional relations. The view definition restrictions, along with restrictions on the sequence order within and across chronicles, guarantees that the views can be maintained incrementally without storing any of the chronicles.

Two recent systems, *OpenCQ* [LPT99] and *NiagaraCQ* [CDTW00], support continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the internet. OpenCQ uses a query processing algorithm based on incremental view maintenance, while NiagaraCQ addresses scalability in number of queries by proposing techniques for grouping continuous queries for efficient evaluation. Within the same project as NiagaraCQ, reference [STD+00] discusses the problem of providing *partial results* to long-running queries on the internet, where it is acceptable to provide an answer over some portion of the input data. The main technical challenge is handling blocking operators in query plans. As will be seen, our architecture provides a framework that captures and classifies all of these issues.

The *Alert* system [SPAM91] provides a mechanism for implementing *event-condition-action* style triggers in a conventional SQL database, by using continuous queries defined over special append-only *active tables*. In Section 4.3.3 we will discuss how Alert and trigger systems in general relate to continuous queries over data streams.

Clearly there is a relationship between continuous queries and the well-known area of *materialized views* [GM95], since materialized views are effectively queries that need to be reevaluated or incrementally updated whenever the base data changes. There are several differences between materialized views and continuous queries: continuous queries may stream rather than store their results, they may deal with append-only input relations, they may provide approximate rather than exact answers, and their processing strategy may adapt as characteristics of the data stream change. Nevertheless, much work on materialized views is captured by our architecture and is relevant to our proposed approach; see Section 4.3.4. Of particularly importance is work on *self-maintenance* [BCL89, GJM96, QGMW96]—ensuring that enough data has been saved to maintain a view even when the base data is unavailable—and the related problem of *data expiration* [GMLY98]—determining when certain base data can be discarded without compromising the ability to maintain a view.

The *Telegraph* project [AH00, HF+00, UF01] shares some target applications and basic technical ideas with our problem, although the general approach is different. Telegraph uses an *adaptive* query engine to process conventional (one-time) queries efficiently under volatile and unpredictable environments (e.g., autonomous data sources over the internet, or sensor networks). The *Tukwila* system [IFF+99] also supports adaptive query processing, in order to perform dynamic data integration over autonomous data sources. Adaptive query processing is likely to be useful for continuous queries over data streams, as discussed in Section 5.

Some work considers traditional data sets but treats them like (finite) data streams, processing the data in a single pass and possibly providing intermediate or "early" query results. For example, *online aggregation* [HHW97,

HH99] is a technique for handling long-running aggregation queries, continually providing a running aggregate with improving probabilistic error bounds. In more theoretical work, [HRR98] studies basic tradeoffs in processing finite data streams, specifically among storage requirements, number of passes required, and result approximations. The problem of computing approximate *quantiles* (equi-height histograms) over numeric data streams of unknown length is addressed in [MRL99] and [GK01].

Recently there has been increasing interest in *data reduction* techniques, where the general goal is to trade accuracy for performance in massive disk-resident data sets, with some obvious possible applications to data streams. A good survey appears in [B+97]. In related work, *synopsis data structures* [GM99] provide a summary of a data set within acceptable levels of accuracy while being much smaller in size, and a framework for extracting synopses (*signatures*) from data streams is proposed in [CFPR00]. A variety of approximate query answering answering techniques have been developed based on data reduction and synopsis techniques including samples [AGPR99, AGP00, CMN99], histograms [IP99, PG99], and wavelets [CGRS00, VW99]. Reference [GKS01] develops histogram-based techniques to provide approximate answers for *correlated aggregate queries* over data streams. Reference [GKMS01] presents a general approach for building small-space summaries over data streams to provide approximate answers for many classes of aggregate queries.

There has been some initial work addressing data streams in the data mining community. In terms of building classical data mining models over a single data stream, reference [Hid99] considers *frequent itemsets* and *association rules*, reference [GMMO00] considers *clustering*, and references [DH00, HSD01] consider *decision trees*. The only work we know of addressing multiple data streams appears in [YSJ+00], which develops algorithms to analyze *co-evolving time sequences* to forecast future values and detect correlations and outliers.

Finally, stream data management and query processing techniques are likely to draw on work in sequence databases (e.g., [SLR94]), time-series databases (e.g., [FRM94]), main-memory databases (e.g., [Tea99]), and real-time databases (e.g., [KGM95]).

3 A Concrete Example

Let us consider a representative application to illustrate the need for continuous queries over data streams and why conventional DBMS technology is inadequate. Consider the domain of *network traffic management* for a large network, e.g., the backbone network of an Internet Service Provider (ISP) [DG00]. Network-traffic-management applications typically process rapid, unpredictable, and continuous data streams, including packet traces and network performance measurements. Due to the inadequacy of conventional DBMSs to provide the kind of online continuous query processing that would be most beneficial in this domain, current traffic-management tools are either restricted to offline query processing or to online processing of simple hard-coded continuous queries, often avoiding the use of a DBMS altogether. A traffic-management system that could provide online processing of ad-hoc continuous queries over data streams would allow network operators to install, remove, and modify appropriate monitoring queries to support effective management of the ISP's network.

As a concrete example, consider an ISP that collects packet traces from two links (among others) in its network. The first link, called the *customer link*, connects the network of a customer to the ISP's network. The second link, called the *backbone link*, connects two routers within the ISP's network. Each packet trace is a continuous stream of packet headers observed on the corresponding link. For simplicity, we assume that a packet header comprises the five fields listed in Figure 1. We use PT_c and PT_b to denote the packet traces collected from the customer and backbone links respectively.

Field name	Description
saddr	IP address of packet sender
daddr	IP address of packet destination
id	Identification number given by sender so that destination can uniquely identify each packet
length	Length of packet
timestamp	Time when packet header was recorded

Figure 1: Record structure of a packet header.

A first simple continuous query (Q_1) computes the load on the backbone link averaged over one minute periods and notifies the network operator if the load exceeds a threshold T. A SQL version of Q_1 using two self-explanatory functions is:

```
Q1: Select   notifyoperator(sum(length))
    From     PTb
    Group By getminute(timestamp)
    Having   sum(length) > T
```

Although Q_1's functionality might be achievable using triggers in a conventional DBMS, performance concerns may dictate special techniques. For instance, if the PT_b stream is coming very fast (e.g., packets in an optical link), the only feasible approach might be to compute an approximate answer to Q_1 by sampling the data, something conventional triggers are certainly not designed for.

A more complex continuous query (Q_2) finds the fraction of traffic on the backbone link coming from the customer network. Q_2 is an example of an ad-hoc continuous query that a network operator might register to check in response to congestion, whether the customer is a likely cause.

```
Q2: (Select   count (*)
     From     PTc As C, PTb As B
     Where    C.saddr = B.saddr and C.daddr = B.daddr
              and C.id = B.id) /
    (Select count (*) From PTb)
```

Q_2 joins streams PT_c and PT_b on their keys to count the number of common packets on the links. Since unbounded intermediate storage could potentially be required for joining two continuous data streams, the network operator might want the system to compute an approximate answer. Possible approximation methods are to allocate a fixed amount of storage and maintain *synopses* of the two streams (recall Section 2), and/or exploit application semantics—such as a high probability that joining tuples occur within a certain time window—to bound the required storage.

A final example continuous query (Q_3) monitors the top 5% source-to-destination pairs in terms of traffic on the backbone link. (We use the SQL3 `With` construct [UW97] for ease of expressing the query.)

```
Q3: With Load As
      (Select     saddr, daddr, sum(length) as traffic
       From       PTb
       Group By saddr, daddr)
    Select       saddr, daddr, traffic
    From         Load As L1
    Where        (Select   count(*)
                  From     Load as L2
                  Where    L2.traffic < L1.traffic) >
                 (Select   0.95×count(*) From Load)
    Order By     traffic
```

Processing Q_3 over the continuous data stream PT_b is especially challenging due its overall complexity and the presence of `Group By` and `Order By` clauses, which are normally "blocking" operators in a query execution plan.

Note that in addition to the issues discussed in each example, all three example queries are likely to benefit from adaptive query processing [AH00], given the unpredictable nature of network packet streams.

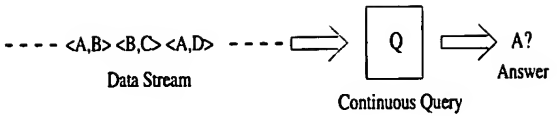


Figure 2: A continuous query Q over a single data stream.

4 Architecture for Continuous Queries

Now that we have seen a concrete example motivating data streams and continuous queries, the remainder of the paper addresses the general problem. We begin in Section 4.1 by motivating, through an extremely simple scenario, some of the most basic issues that arise when processing continuous queries over data streams. Then in Section 4.2 we present our architecture, which allows us in Section 4.3 to classify previous work in continuous queries, and to relate continuous queries to triggers and materialized views. We consider data streams that adhere to the relational model (i.e., streams of tuples), although many of the ideas and techniques are independent of the data model being considered.

4.1 Motivation

Let us consider the simplest possible scenario to illustrate the differences between querying data streams and traditional stored data sets. Suppose we have a single, continuous stream of tuples and a single query Q we are interested in answering over the stream, as illustrated in Figure 2. Q is a continuous query—we issue it once and it operates continuously as new tuples appear in the stream—and suppose we are interested in the exact answer to Q (as opposed to an approximation). Let us further suppose that the data stream is *append-only*—it has no updates or deletions—so we can think of the stream as an unbounded append-only database D. Even in this simplest of cases, there are different possible ways to handle Q, with different ramifications:

(1) Suppose we want to always store and make available the current answer A to Q. Since the "database" D may be of unbounded size, the size of A also may be unbounded (e.g., if Q is a selection query).

(2) Suppose instead we choose not to store answer A, but rather to make new tuples in A available when they occur, e.g., as another continuous data stream. Although we no longer need unbounded storage for A, we still may need unbounded storage for keeping track of tuples in the data stream in order to determine new tuples in A (e.g., if Q is a self-join).

Let us further complicate the problem by considering deletions and updates:

(3) Even if the stream is append-only, there may be updates or deletions to tuples in answer A (e.g., if Q is a group-by query with aggregation). Now, in case (2) above we may need to somehow update and delete tuples in our output data stream, in addition to generating new ones.

(4) In the most general scenario, the input data stream also may contain updates or deletions. In this case, typically more—possibly much more—of the stream needs to be stored in order to continuously determine the exact answer to Q.

One way to address these issues is to restrict the expressiveness of Q and/or impose constraints on characteristics of the data stream so that we can guarantee that the size of Q's answer A is bounded, or that the amount of extra storage needed to continuously compute A is bounded. Previous work on continuous queries, e.g., [JMS95, TGNO92, Bar99], has tended to take this approach. Another possibility is to relax the requirement that we always provide an exact answer to Q, which relates to the area of *approximate query answering* discussed in Sections 2 and 3.

In this paper we do not specifically advocate one of these approaches. Instead, we specify a general and flexible architecture that makes the choices above, and their ramifications, explicit. We further use our basic architecture to explain how continuous queries relate to triggers and materialized views, and to define a number of open research problems in processing continuous queries over data streams.

4.2 Architecture

We now introduce our general architecture for processing continuous queries over data streams, illustrated in Figure 3. For now let us consider a single continuous query Q with answer A, operating over any number of incoming data streams. Multiple continuous queries can be handled within our architecture (as implied in the figure), and we will discuss some of the interesting issues that arise in this context in Section 5.4. We also assume that the query is over data streams only, although mixing streams and conventional relations poses no particular problems.

When query Q is notified of a new tuple t in a relevant data stream, it can perform a number of actions, which are not mutually exclusive:

(i) It can determine that because of t there are new tuples in the answer A. If it is known that a new tuple a in A will remain in A "forever," then Q may send tuple a to the *Stream* component illustrated in Figure 3.

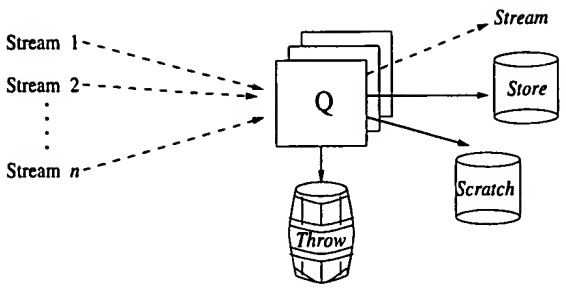


Figure 3: Architecture for processing continuous queries over data streams.

In other words, *Stream* is a data stream containing tuples appended to A, similar to case (2) discussed in Section 4.1.

(ii) If a new tuple a is determined to be in A, but may at some time no longer be in A, then a is added to the *Store* component illustrated in Figure 3. In other words, together *Stream* and *Store* define the current query answer A. If our goal is to minimize storage for the query result, then we want to make sure that tuples are sent to *Stream* rather than *Store* whenever possible.

(iii) The new stream tuple t may cause the update or deletion of answer tuples in *Store*. Answer tuples might also be moved from *Store* to *Stream*.

(iv) We may need to save t, or save data derived from t, so that in the future we are assured of being able to compute our query result. In this case, t (or the data derived from it), is sent to the *Scratch* component of Figure 3. Combined with action (iii), we might also move data from *Store* to *Scratch*.

(v) We may not need t now or later, in which case t is sent to the *Throw* component of Figure 3. Note that *Throw* does not require any actual storage (unless we are interested in archiving unneeded data).

(vi) As a result of the new stream tuple t, we may take data previously saved in *Scratch* (or *Store*) and send it to *Throw* instead. If our goal is to minimize storage, we want to make sure that unneeded data is sent to *Throw* whenever possible, rather than *Scratch*.

4.3 The Architecture and Related Work

In this section we revisit the issues and scenarios discussed in Section 4.1, revisit the related work discussed in Section 2, and consider triggers and materialized views. In all cases we use our basic architecture as a tool for detailed understanding and comparisons.

4.3.1 Query Processing Scenarios

Let us consider query processing scenarios (1)–(4) from Section 4.1 in light of the architecture specified in Section 4.2. In scenario (1), we want to always store Q's entire current answer A. In terms of our architecture, (1) says that *Stream* is empty, *Store* always contains A, and *Scratch* contains any data that may be required to keep the answer in *Store* up-to-date. In the example case where Q is a selection query, *Store* may be of unbounded size, while *Scratch* is empty. Conversely in scenario (2) we want to make A available exclusively as a data stream, i.e., *Stream* streams the entire answer to A while *Store* is empty. In the example case where Q is a self-join, we can send all answer tuples to *Stream* since they will remain in the result forever, but *Scratch* may need to grow without bound.

Scenario (3) covers the case where answer A can have updates and deletions even when the input streams are append-only, e.g., a query that performs grouping and aggregation. Scenario (4) further extends to the case where the input streams may have updates and deletions. As an example, suppose Q is a group-by query over a single data stream with a *min* aggregation function. Since *min* is monotonic for insertions, in scenario (3) A is maintained in *Store*, and *Scratch* can remain empty. However, in scenario (4) unbounded storage is required for *Scratch* to ensure that the *min* values over the entire stream can always be computed. In both cases, the only time answer tuples can be sent to *Stream*, or moved from *Store* to *Stream*, is when it is known that for some group there will be no further insertions, updates, or deletions of tuples falling into that group.[1]

4.3.2 Previous Related Work

We now revisit some of the related work discussed in Section 2, characterizing it in terms of our basic architecture. Note that citations are not repeated in this section except when needed to identify the work being discussed. Also note that some of the related work from Section 2 is revisited instead in Section 4.3.3 on triggers or Section 4.3.4 on views.

Recall that the Tapestry system supports restricted continuous queries over append-only data sets. In Tapestry, a continuous query Q is rewritten into its *minimum bounding monotone query* Q^M, which is then rewritten into an *incremental query* Q^I. As a monotone continuous query, Q^M has the property that its answer changes only by addition of new tuples, so in terms of our architecture all answer tuples can be sent to *Stream* and *Store* is empty. The incremental version Q^I of the query is meant to improve the efficiency of computing new answer tuples when new input tuples are appended, but there is no mechanism for guaranteeing that *Scratch* will not grow without bound.

The work in [STD+00] on maintaining partial results for long-running queries is similar to Scenario (3) in Section 4.1. It maintains the current partial result in *Store* and any extra needed information in *Scratch*. Our discussion of new query processing techniques in Section 5.3 is relevant to the problem addressed in [STD+00], and we believe that based on these techniques it is possible to exploit monotonicity more aggressively to improve upon the algorithm in [STD+00], reducing the data saved in *Scratch*. *OpenCQ* and *NiagaraCQ* consider Scenario (4) in Section 4.1, but they are geared towards data sets that change primarily through in-place updates. Thus, they do not address the problem of *Store* or *Scratch* growing without bound.

A number of systems perform tuple-at-a-time processing over their input data streams: each time a new stream element arrives, the element is moved directly to either *Stream* or *Throw*, without consulting any other data in the stream. Packet routing and simple network algorithms have this characteristic [Tan96], although for network traffic management more sophisticated stream processing is needed, as seen in Section 3. The *XFilter* and *Xyleme* systems discussed in Section 2 also perform element-at-a-time processing although the elements are XML documents.

Basic online aggregation [HHW97] maintains the current aggregate in *Store* along with an estimate of the error, and an empty *Scratch*. Follow-on work that extends online aggregation to joins [HH99] does need to maintain previously seen tuples in *Scratch*. Finally, the body of work in approximate query answering focuses primarily on making the best possible use of a limited size *Scratch* by storing only small synopses (summaries) of the data. References [GMP97, MRL99, MVW00, Vit85] address the problem of updating the synopses (i.e., *Scratch*) efficiently when the underlying data changes.

4.3.3 Triggers

Triggers, also called *event-condition-action* rules, are used to monitor events and conditions in databases, and to execute actions automatically when specific situations are detected [WC96]. In the Alert system introduced in Section 2, triggers are implemented by means of continuous queries over *active tables*. Each tuple in an active table represents an *event*, which is an update on a conventional stored table. When a new tuple is added to one of the active tables, each continuous query involving the ta-

[1] Note that we are assuming *Stream* is constrained to be append-only, even though in scenario (4) we discuss input streams with updates and deletions. If we allow updates and deletions to *Stream* tuples, then we are always free to send answer tuples to *Stream* instead of *Store*, since we can update or delete them later.

ble is evaluated, and the trigger *action* is invoked on each new tuple in the query result.

Our mapping from triggers to the architecture of Figure 3 is based on (and slightly generalizes) the Alert approach. We assume that events to be monitored are generated as data streams, and we allow continuous queries over any number of data streams together with conventional stored tables. As in Alert, these queries perform event and condition monitoring. For launching trigger actions, like Alert we assume that the desired actions are performed by SQL data manipulation commands and user-defined stored procedures specified as part of the continuous queries (e.g., query Q_1 in Section 3). In terms of our architecture, since there is no query "answer" in triggers, *Stream* and *Store* may remain empty, while *Scratch* is used for any data required to monitor complex events or evaluate conditions. Alternatively, depending on the desired trigger behavior and application interaction, actions could send results to *Stream*.

There are a number of benefits to using continuous queries over data streams to provide trigger functionality. Continuous queries specified on event streams together with conventional tables enable complex multi-table events and conditions to be monitored, equivalent to the most powerful trigger language proposals we know of [WC96]. More importantly, trigger processing would benefit automatically from efficient data management and processing techniques for continuous queries over data streams, such as specialized query optimization techniques (Section 5.3).

4.3.4 Materialized Views

Materialized views, whether in a conventional DBMS or in a *data warehousing* environment [GM95], fall naturally into our architecture. The base data over which the views are defined, if not available in conventional stored tables, is stored in *Scratch*. The view itself is maintained in *Store*. Updates to the base data can be represented as one or more data streams, as discussed in Section 4.3.3 for triggers. In terms of this mapping, work on materialized view self-maintenance and expiration, discussed in Section 2, is geared specifically towards minimizing the size of *Scratch*. Pure self-maintenance guarantees that *Scratch* is empty [BCL89, GJM96], although for many views pure self-maintainability is impossible, so *auxiliary views* must be stored and maintained in *Scratch* [QGMW96]. Data expiration exploits constraints to determine precisely when data can be removed from *Scratch*, although no bounds on the size of *Scratch* are guaranteed. The Chronicle data model discussed in Section 2 for materialized views is designed to ensure bounded storage for *Scratch*, but like pure self-maintainability it restricts the allowable view definitions significantly. To the best of our knowledge, no work on materialized views has addressed the problem of bounding the size of the materialized view itself, so that the size of *Store* also can be bounded.

5 Research Problems

In this section we outline a number of research problems associated with processing continuous queries over data streams. We begin at a relatively global level, becoming more detailed as the section progresses. In several cases the architecture of Section 4.2 is used to make the problems and issues more concrete.

5.1 Basic Problems and Techniques

At the most global level, what sets continuous queries over data streams apart from previous work is a unique combination of:

- **Online processing.** The applications discussed in Section 1 require that continuous queries are processed, well, continuously. Specifically, when new tuples arrive in a data stream they generally must be "consumed" immediately, usually performing one or more of actions (i)–(vi) from Section 4.2. In some applications the tuples may arrive so fast that some of them need to be ignored entirely.

- **Storage constraints.** In the general case for continuous data streams, the amount of storage required for the answer to a continuous query, or to ensure that the answer always can be computed, may be unbounded (recall Section 4.1). Furthermore, even if there is "nearly" unbounded storage available on disk or other tertiary devices, performance requirements may be such that *Store* and/or *Scratch* from Figure 3 need to reside in a limited amount of main memory.

While neither of these problems in isolation is entirely new, dealing with them together, while at the same time offering the full functionality and efficiency of a database query processor, is a new challenge.

Next we mention three basic techniques that have been explored primarily in other contexts within the database or broader Computer Science research community. All of them appear directly relevant to our problem.

- **Summarization.** *Summaries* (or data *synopses*) provide a concise representation of a data set at the expense of some accuracy. As discussed in Section 2, many techniques for summarization have been developed, including *sampling*, *histograms*, and *wavelets*. (See Section 2 for citations.) We expect summarization to play an important role in query processing over data streams due to the storage constraints

discussed above. New issues to resolve in the data stream environment include: (i) how to make guarantees about accuracy of continuous query results based on summaries; (ii) how to maintain summaries efficiently in the presence of very rapid data streams; (iii) what summarization techniques are best for unpredictable data streams. We revisit some of these issues in Section 5.3.

- **Online data structures.** A data structure designed specifically to handle continuous data-flow is typically referred to as an *online data structure* [FW98]. Continuous queries by nature suggest the use of online data structures for query processing.

- **Adaptivity.** We expect continuous queries and the data streams on which they operate to be *long-running*. Unlike during the processing of a simple one-time query, during the lifetime of a continuous query parameters such as the amount of available memory, stream data characteristics, and stream flow rates may vary considerably. While adaptive query processing techniques for more traditional queries have attracted interest recently (see Section 2 for a discussion), the work so far that we are aware of has not considered all of the parameters or kinds of adaptivity (e.g., changing approximations) that arise in a data stream context.

Distilling the basic problems and techniques above, we see that processing continuous queries over data streams entails making fundamental tradeoffs among *efficiency*, *accuracy*, and *storage*. References [AMS96, HRR98] provide some initial contributions from the theory community along these lines, but it is an open problem to understand the implications of these tradeoffs in a real system processing continuous queries for one or more real applications.

Next we will consider in more detail several specific research challenges. We will start in Section 5.2 by briefly discussing the issue of languages for specifying continuous queries. Then in Section 5.3 we focus on query evaluation and optimization, including execution plans and operators for continuous queries. We briefly address research problems associated with multiple continuous queries in Section 5.4.

5.2 Languages for Continuous Queries

Although we certainly do not advocate inventing a new query language for the purpose of specifying continuous queries over data streams—particularly over streams of relational tuples—there are some issues that must be considered. Let us take SQL as an example. Most previous work on continuous queries has restricted the language being considered in order to guarantee certain properties such as bounding the size of *Scratch* (or eliminating it entirely), or ensuring that all query results can be sent to *Stream* and none to *Store*. It appears to be an open problem to determine for arbitrary SQL queries whether these kinds of properties are satisfied, particularly if we accept the use of *Scratch* and *Store* but want to make sure they are bounded in some way. We also believe that for certain applications continuous queries will need to refer to the sequencing aspect of streams. Here SQL with extensions for *ordered relations* [SLR94], or with built-in *time-series* support [FRM94], might be a reasonable choice.

5.3 Query Evaluation and Optimization

In any database system it is the job of the query optimizer to choose in advance the "best" query plan for executing each query, based on a variety of statistics maintained for this purpose. A continuous query processor also should select a "best" execution plan, although we expect that fewer of the decisions will be made in advance due to the long-running nature of continuous queries discussed in Section 5.1. Techniques such as *eddies* [AH00], which construct and adapt query plans on-the-fly, come the closest that we know of to the query execution style we envision. However, that work is still designed for one-time rather than continuous queries, the query execution strategies do not adapt to all relevant parameters in the data stream context, and the notion of adaptivity is geared solely towards online processing.

Let us assume a standard *pipelined* (or *iterator-based*) approach to query processing [Gra93]. One of the fundamental differences between traditional query plans operating over stored relations and plans operating over data streams can be characterized as "push" versus "pull." Specifically, a traditional query plan usually has a tree shape and is executed top-down in a "pull" style: each query operator polls its children for the required input, ultimately accessing stored indexes or relations at the leaves of the query tree. Parallel query plans relax this paradigm to some extent [Gra90], but usually do not use the fully "push-based" model that data streams may demand. In an execution plan for a continuous query over data streams, we expect that it will be the appearance of a new tuple in a relevant stream that sets the plan into action. Of course this idea is not new, but rather a query processing variant on triggers, alerts, and other "active" constructs in databases [WC96].

"Push" versus "pull" aside, let us consider other changes that may be required to adapt traditional query plan operators to the data stream context. We will first consider true pipelined operators (such as selections and joins), then we will consider *blocking* operators (such as aggregation and sorting). Finally we will consider a

new class of operators that may be useful for continuous queries over data streams.

5.3.1 Pipelined operators

The simplest standard pipelined operators, such as selections, can be translated to the data stream context with little modification. However, as soon as we introduce joins we are faced with a choice. We can either: (i) evaluate portions of the query multiple times as in a nested-loop style join, which we assume is undesirable or even impossible in the data stream context; or (ii) use *Scratch* to hold temporary results during query processing, as in a pipelined hash join [WA91].

The case of joins points out that when processing continuous queries over data streams, we not only want our query operators to be pipelined, we also want them to operate with bounded intermediate storage (even in the presence of unbounded streams). For example, we might modify a pipelined join operator to degrade gracefully to an approximate join when the required storage begins to reach limits. Semantic constraints in the spirit of *data expiration* [GMLY98], or online feedback across operators in the spirit of *ripple joins* [HH99], could be applied to compute approximations with minimal loss of information.

As it turns out, the architecture we introduced in Section 4.2 for continuous queries as a whole also applies nicely to individual query plan operators: *Store* and *Scratch* represent the intermediate storage required by an operator, while *Stream* represents the pipelined operator results. Thus, techniques developed at the query level for summarization, approximation, or for moving data from *Scratch* or *Store* to *Stream* or *Throw*, might be applicable recursively to query plan operators. It is important to bear in mind, however, that *Scratch* and *Store* will generally be bounded globally, not on a per-operator basis.

5.3.2 Blocking Operators

A *blocking* operator is one that must obtain its entire input set before it can produce any output—typical examples are sorting and aggregation. In a conventional pipelined query plan, all operators that follow a blocking operator must wait until the operator obtains its entire input and begins producing its results. Obviously blocking operators cannot behave in their conventional fashion in the presence of continuous data streams, since the input is unbounded and the operator would block "forever." Part of the solution to this problem must be based on semantic considerations such as those discussed in Section 4.1—e.g., what is the result of an aggregation or a sort now when more data may be coming later? In addition to techniques such as online aggregation [HHW97, HH99], there has been some work addressing closely-related problems [LPT99, STD+00] that develops techniques based on incremental view maintenance. Developing similar techniques for continuous queries over data streams, and even more fundamentally understanding the semantics implied by the various techniques, remains an open problem.

5.3.3 Synopsis Operators

We discussed the requirement for summaries or synopses in Section 5.1 and cited some of the most relevant work in Section 2. One approach to incorporating synopsis data structures into a database system is to encapsulate them as basic operators that may appear in query plans. In support of this approach, reference [GM99] shows that different classes of queries are supported efficiently by different synopsis data structures. Thus, the query optimizer could be charged with choosing the best synopsis operator for each purpose under current conditions.

Taking this idea one step further, synopsis query operators could provide the capability to "tune" certain parameters within the operator, such as accuracy and confidence of approximation (e.g., probabilistic confidence bounds for aggregates [HHW97]), and maximum storage required (e.g., a random sample of size N). Particularly relevant in this context are the *semantic synopsis structures* proposed in [BGR01], which summarize a massive disk-resident relation based on error tolerance parameters provided independently for each attribute. If we provide synopsis operators with these types of parameters, then approximate query plans can be constructed carefully based on the query structure and available storage. Of course this power also poses significant challenges for the query optimizer.

5.4 Multiple Continuous Queries

In the paper so far we have assumed a single continuous query over multiple data streams. Let us now consider the more realistic scenario where an application registers multiple continuous queries simultaneously, probably over shared data streams. Because continuous queries are long-running, and some applications may involve a very large number of continuous queries, we expect that some form of *multi-query optimization* [Fin82, Sel88, CDTW00] will be a relevant and perhaps essential technique. There has been some recent work on optimizing multiple continuous queries, focusing either on very large numbers of queries where each query performs element-at-a-time processing [AF00, NACP01], or on subquery merging in the XML context [CDTW00]. In terms of our architecture, the queries in these systems are limited enough that they

always have empty or bounded *Store* and *Scratch* components.

Research yet to be performed includes extending the techniques from [AF00, NACP01, CDTW00] to handle more complex queries, coupling multi-query optimization techniques with approximate query answering, and optimizing the use of bounded-size *Scratch* and *Store* when they are shared among many continuous queries. More generally, the overall problem of understanding and implementing the tradeoffs among efficiency, accuracy, and storage becomes at least one step more complex in the presence of multiple continuous queries.

6 A Data Stream Management System

Our ultimate goal is to build a complete *data stream management system* (*DSMS*), with functionality and performance similar to that of a traditional DBMS, but which allows some or all of the data being managed to come in the form of continuous, possibly very rapid, data streams. In such a system, traditional one-time queries are replaced or augmented with continuous queries, and techniques such as synopsis and online data structures, approximate results, and adaptive query processing become fundamental features of the system. Other aspects of a complete DBMS also need to be reconsidered, including storage management, transaction management, user and application interfaces, and authorization.

Obviously building a complete DSMS—even a research prototype—entails a significant effort. One approach would be to modify or extend an existing DBMS to include the functionality that we envision. However, our approach will be to build a complete DSMS from scratch, so we can fully explore the issues under our own control. We have described many novel and interesting research problems that we expect to encounter along the way.

7 Conclusions and Research Plan

Many recent applications need to process continuous data streams in addition to or instead of conventional stored data sets. In this paper we have specified a general and flexible architecture for processing *continuous queries* in the presence of data streams. We have used our basic architecture as a tool to clarify alternative semantics and processing techniques for continuous queries, as well as to relate past and current work to the general *Data Stream Management System* (*DSMS*) we envision. We have mapped out a number of research topics in the area of query processing over data streams, including new requirements for online, approximate, and adaptive query processing.

At Stanford we have begun to build a complete prototype DSMS called *STREAM* (*STanford stREam datA Manager*). We are focusing initially on:

- A flexible interface for reading and storing data streams—or stream synopses—as part of a hierarchical storage manager.
- A processor for continuous queries specified using SQL or relational algebra including aggregation.
- A client Application Programming Interface (API) for registering continuous queries and receiving query results.

We expect that the development of our prototype system, as well as continuous detailed evaluation of potential applications such as the network monitoring system described in Section 3, will lead to further algorithmic and system research issues. Please visit `http://www-db.stanford.edu/stream`.

Acknowledgements

We are grateful to Jose Blakeley for excellent comments on an initial draft, and to the entire STREAM group at Stanford for many inspiring discussions.

References

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 53–64, September 2000.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 487–498, May 2000.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, June 1999.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the 1996 Annual ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[B+97] D. Barbara et al. The New Jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[Bar99] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, December 1999.

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[BGR01] S. Babu, M. N. Garofalakis, and R. Rastogi. SPARTAN: A model-based semantic compression system for massive data tables. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 283–294, May 2001.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[CFPR00] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 9–17, August 2000.

[CGRS00] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 111–122, September 2000.

[CMN99] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 1999.

[DG00] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. of the 2000 ACM SIGCOMM*, pages 271–284, September 2000.

[DH00] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 71–80, August 2000.

[Fin82] S. J. Finkelstein. Common subexpression analysis in database applications. In *Proc. of the 1982 ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, June 1982.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–429, May 1994.

[FW98] A. Fiat and G. J. Woeginger. *Online Algorithms, The State of the Art*. Springer-Verlag, Berlin, 1998.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of the 1996 Intl. Conf. on Extending Database Technology*, pages 140–144, March 1996.

[GK01] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 58–66, May 2001.

[GKMS01] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, September 2001.

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, May 2001.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GM99] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *External Memory Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 50, 1999.

[GMLY98] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 500–511, August 1998.

[GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. of the 2000 Annual Symp. on Foundations of Computer Science*, pages 359–366, November 2000.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 466–475, August 1997.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pages 102–111, May 1990.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HF+00] J. M. Hellerstein, M. J. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[HH99] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 287–298, June 1999.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, May 1997.

[Hid99] C. Hidber. Online association rule mining. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 145–156, June 1999.

[HRR98] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report TR-1998-011, Compaq Systems Research Center, Palo Alto, California, May 1998.

[HSD01] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. of the 2001 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, August 2001. (To appear).

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data

integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, pages 174–185, September 1999.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.

[KGM95] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 463–486. Prentice Hall, 1995.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):583–590, August 1999.

[MRL99] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 251–262, June 1999.

[MVW00] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 101–110, September 2000.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 437–448, May 2001.

[PG99] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *Proc. of the 1999 Intl. Conf. on Scientific and Statistical Database Management*, pages 24–33, July 1999.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, 1988.

[SLR94] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 430–441, May 1994.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pages 469–478, September 1991.

[STD+00] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *Proc. of the 2000 Intl. Workshop on the Web and Databases*, pages 17–22, May 2000.

[Sul96] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, page 594, September 1996.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Tea99] Times-Ten Team. In-memory data management for consumer transactions: The Times-Ten approach. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 528–529, June 1999.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[Tra] Traderbot home page. http://www.traderbot.com.

[UF01] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, September 2001. (To appear).

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

[Vit85] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, March 1985.

[VW99] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 193–204, June 1999.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the 1991 Intl. Conf. on Parallel and Distributed Information Systems*, pages 68–77, December 1991.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.

[XPA99] XML path language (XPath) version 1.0, November 1999. W3C Recommendation available at http://www.w3.org/TR/xpath.

[YSJ+00] B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 13–22, March 2000.

Maintaining Stream Statistics over Sliding Windows

Mayur Datar* Aristides Gionis† Piotr Indyk‡ Rajeev Motwani§

July 30, 2001

Abstract

We consider the problem of maintaining aggregates and statistics over data streams, with respect to the last N data elements seen so far. We refer to this model as the *sliding window* model. We consider the following basic problem: Given a stream of bits, maintain a count of the number of 1's in the last N elements seen from the stream. We show that using $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory, we can estimate the number of 1's to within a factor of $1+\epsilon$. We also give a matching lower bound of $\Omega(\frac{1}{\epsilon}\log^2 N)$ memory bits for any deterministic or randomized algorithms. We extend our scheme to maintain the sum of the last N positive integers. We provide matching upper and lower bounds for this more general problem as well. We apply our techniques to obtain efficient algorithms for the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model. Using the algorithm for the basic counting problem, one can adapt many other techniques to work for the sliding window model, with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. These include maintaining approximate histograms, hash tables, and statistics or aggregates such as sum and averages.

*Department of Computer Science, Stanford University, Stanford, CA 94305. Supported by Microsoft Graduate Fellowship. Email:datar@cs.stanford.edu.

†Department of Computer Science, Stanford University, Stanford, CA 94305. Email:gionis@cs.stanford.edu.

‡MIT Laboratory for Computer Science, 545 Technology Square, NE43-373, Cambridge, Massachusetts 02139. Email:indyk@theory.lcs.mit.edu

§Department of Computer Science, Stanford University, Stanford, CA 94305. Email:rajeev@cs.stanford.edu

1 Introduction

For many recent applications, the concept of a *data stream*, possibly infinite, is more appropriate than a data set. By nature, a stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times. One of the challenging aspects of processing over data streams is that while the length of a data stream may be unbounded, making it impractical or undesirable to store the entire contents of the stream, for many applications[1] it is still important to retain some ability to execute queries that reference past data. In order to support queries of this sort using a bounded amount of storage, it is necessary to devise techniques for storing summary or synopsis information about previously seen portions of data streams. Generally there is a tradeoff between the size of the summaries and the ability to provide precise answers to queries involving past data.

We consider the problem of maintaining statistics over streams with regard to the last N data elements seen so far. We refer to this model as the *sliding window* model. We identify a simple counting problem whose solution is a prerequisite for efficient maintenance of a variety of more complex statistical aggregates: Given a stream of bits, maintain a count of the number of 1's in the last N elements seen from the stream. We show that using $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory, we can estimate the number of 1's to within a factor of $1+\epsilon$. We also give a matching lower bound of $\Omega(\frac{1}{\epsilon}\log^2 N)$ memory bits for any deterministic or randomized algorithm.

We extend our scheme to maintain the sum of the last N positive integers. We provide matching upper and lower bounds for this more general problem as well. We apply our techniques to obtain efficiently algorithms for the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model. Using the algorithm for the basic counting problem, one can adapt many other techniques to work for the sliding window model, with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. These include maintaining approximate histograms, hash tables, and statistics or aggregates such as sum and averages.

1.1 Motivation, Model, and Related Work

Several applications naturally generate data streams. In telecommunications, for example, *call records* are generated continuously. Typically, most processing is done by examining a call record once, or operating on a "window" of recent call records (e.g., to update customer billing information), after which records are archived and not examined again. Cortes et al [2] report working with AT&T long distance call records, consisting of 300 million records per day for a 100 million customers. A second application is *network traffic engineering*, where information about current network performance (e.g., latency and bandwidth) is generated online and is used to monitor and adjust network performance dynamically [7, 14]. It is generally both impractical and unnecessary to process anything but the most recent data.

There are other traditional and emerging applications where data streams play an important and natural role, e.g., web tracking and personalization (the streams are web log entries or click-streams), medical monitoring (vital signs, treatments, and other measurements), sensor databases, and financial monitoring, to name but a few. There are also applications where traditional (non-streaming) data is treated as a stream due to performance constraints. In data mining applications,

[1] For example, in order to detect fraudulent credit card transactions, it is useful to be able to detect when the pattern of recent transactions for a particular account differs significantly from the earlier transactional history of that account.

for example, the volume of data stored on disk is so large that it is only possible to make one pass (or perhaps a very small number of passes) over the data [10, 9]. The objective is to perform the required computations using the stream generated by a single scan of the data, using only a bounded amount of memory and without recourse to indexes, hash tables, or other precomputed summaries of the data. Another example is that data streams are generated as intermediate results of pipelined operators during evaluation of a query plan in an SQL database—without materializing some or all of the temporary result, only one pass on the data is possible [3].

In most applications, the goal is to make decisions based on the statistics or models gathered over the "recently observed" data elements. For example, one might be interested in gathering statistics about packets processed by a set of routers over the last day. Moreover, we would like to maintain these statistics in a continuous fashion. This gives rise to the *sliding window model:* Data elements arrive at each instant and expire after exactly N time steps; and, the portion of data that is relevant to gathering statistics or answering queries is set of the last N elements to arrive. The sliding window refers to the window of active data elements at any time instant.

Previous work [1, 5, 11] on stream computations addresses the problems of approximating frequency moments and computing the L_p differences of streams. There has also been work on maintaining histograms [12, 8]. While Jagadish et al [12] address the off-line version of computing optimal histograms, Guha and Koudas [8] give a technique for maintaining near optimal time-based histograms over streaming data. The queries that are supported by histograms constructed in the latter work are range or point queries over the time attribute. In the earlier work, the underlying model is that all the data elements seen so far are relevant. We believe that the sliding window model is perhaps more important since for most applications one is not interested in gathering statistics over outdated data. Maintaining statistics like sum/average, histograms, hash tables, frequency moments, and L_p differences over sliding windows is critical to most applications. To our knowledge, there is no previous work addressing these problems for the sliding window model.

1.2 Summary of Results

We focus on the sliding window model for data streams. We formulate a basic counting problem whose solution can be used as a building block for solving most of the problems mentioned earlier.

Problem 1 (BASICCOUNTING) *Given a stream of data elements, consisting of* 0*'s and* 1*'s, maintain at every time instant the count of the number of* 1*'s in the last* N *elements.*

It is easy to verify that an exact solution requires $\Theta(N)$ bits[2] of memory. For most applications it is prohibitive to use $\Omega(N)$ memory. For instance, consider the network management application where a large number of data packets pass through a router every second. However, in most applications it suffices to produce an approximate answer. Thus, our goal is to provide a good approximation using $o(N)$ memory.

In Section 2, we provide a solution for BASICCOUNTING which uses $O(\frac{1}{\epsilon}\log^2 N)$ bits of memory (equivalently $O(\frac{1}{\epsilon}\log N)$ buckets of size $O(\log N)$) and provides an estimate of the answer at every instant that is within $1+\epsilon$ factor of the actual answer. Moreover, our algorithm does not require an a priori knowledge of N and caters to the possibility that the window size can be changed dynamically. Our algorithm is guaranteed to work with $O(\log^2 N)$ memory as long as the window size is bounded by N. The algorithm takes $O(\log N)$ worst-case time to process each new data element's arrival, but only $O(1)$ amortized time per element. Count queries can be processed in $O(1)$ time. The algorithm itself is relatively simple and easy to implement.

[2]Note that we measure space complexity in terms of number of bits, rather than number of memory words.

Section 3 presents a matching lower bound. We show that any approximation algorithm (deterministic or randomized) for BASICCOUNTING with with relative error $1+\epsilon$ must use $\Omega(\frac{1}{\epsilon}\log^2 N)$ bits of memory. This proves that our algorithm is optimal in terms of memory usage.

In Section 4 we extend the technique to handle data elements with positive integer values, instead of just binary values, referred to as the SUM problem. We provide matching upper and lower bounds on the memory usage for this general problem as well. Then, in Section 5 we show we can use our techniques along with the sketching techniques of Indyk [11] to efficiently maintain the L_p norms (for $p \in [1,2]$) of vectors under the sliding window model.

Section 6 provides a brief discussion of the application of the BASICCOUNTING and SUM algorithms to adapting several other problems in the sliding window model, such as maintaining histograms, hash tables, and statistics or aggregates such as averages/sums. The reduction of these problems to BASICCOUNTING entails a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1+\epsilon$ factor loss in accuracy. We also discuss problems such as Min/Max and Distinct Values.

2 Algorithm for BASICCOUNTING

Our approach towards solving the BASICCOUNTING problem is to maintain a histogram that records the timestamp of selected 1's that are *active* in that they belong to the last N elements. We call this histogram the Exponential Histogram (EH) for reasons that will be clear later. Before getting into the details of our algorithms we need to introduce some notation.

We follow the conventions illustrated in Figure 1. In particular, we assume that new data elements are coming from the right and the elements at the left are ones already seen. Note that each data element has an *arrival time* which increments by one at each arrival, with the leftmost element considered to have arrived at time 1. But, in addition, we employ the notion of a *timestamp* which corresponds to the position of an *active* data element in the current window. We timestamp the active data elements from right to left, with the most recent element being at position 1. Clearly, the timestamps change with every new arrival and we do not wish to make explicit updates. A simple solution is to record the arrival times in a wraparound counter of $\log N$ bits and then the timestamp can be extracted by comparison with counter value of the current arrival. As mentioned earlier, we concentrate on the 1's in the data stream. When we refer to the k-th 1, we mean the k-th most recent 1 encountered in the data stream.

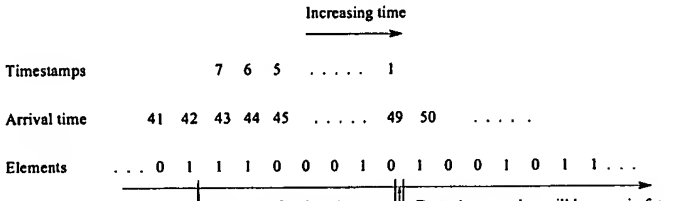


Figure 1: An illustration for the notation and conventions followed.

We will maintain histograms for the active 1's in the data stream. For every bucket in the histogram, we keep the timestamp of the most recent 1 (called *timestamp*), and the number of 1's (called *bucket size*). When the timestamp of a bucket expires (reaches $N+1$), we are no longer interested in data elements contained in it, so we drop that bucket and reclaim its memory. If a

bucket is still active, we are guaranteed that it contains at least a single 1 that has not expired. Thus, at any instant there is at most one bucket (the last bucket) containing 1's which may have expired. At any time instant we may produce an estimate of the number of active 1's as follows. For all but the last bucket, we add the number of 1's that are there in them. For the last bucket, let C be the count of the number of 1's in that bucket. The actual number of active 1's in this bucket could be anywhere between 1 and C, so we estimate it to be $C/2$. We obtain the following:

Fact 1 *The absolute error in our estimate is at most $C/2$, where C is the size of the last bucket.*

Note that for this approach the window size does not have to be fixed a-priori at N. Given a window size S, we calculate the expiry time and do the same thing as before except that the last bucket is the bucket with the largest timestamp that is less than the expire time.

2.1 The Approximation Scheme

We now define the Exponential Histograms and present a technique to maintain them, so as to guarantee count estimates with relative error at most ϵ, for any $\epsilon > 0$. Define $k = \lceil \frac{1}{\epsilon} \rceil$, and assume that $\frac{k}{2}$ is an integer; if $\frac{k}{2}$ is not an integer we can replace $\frac{k}{2}$ by $\lceil \frac{k}{2} \rceil$ without affecting the basic results.

As per Fact 1, the absolute error in the estimate is $C/2$, where C is the size of the last bucket. If C_i is the size of the i-th bucket, we know that the true count is at least $1 + \sum_{i=1}^{m-1} C_i$, since the last bucket contains at least one 1 and the remaining buckets contribute exactly their size to total count. Thus, the relative estimation error at most $C_m/2(1 + \sum_{i=1}^{m-1} C_i)$. We will ensure that the relative error is at most $1/k$ by maintaining the following invariant:

Invariant 1 *At all times, the bucket sizes $C_1, \ldots, C_m$ are such that: For all $j \leq m$, we have $C_j/2(1 + \sum_{i=1}^{j-1} C_i) \leq \frac{1}{k}$.*

Let $N' \leq N$ be the number of 1's that are active at any instant. Then the bucket sizes must satisfy $\sum_{i=1}^{m} C_i \geq N'$. In order to satisfy this and Invariant 1 with as few buckets as possible, we maintain buckets with exponentially increasing sizes so as to satisfy the following second invariant.

Invariant 2 *At all times the bucket sizes are nondecreasing, i.e., $C_1 \leq C_2 \leq \cdots \leq C_{m-1} \leq C_m$. Further, the bucket sizes are constrained to the following: $\{1, 2, 4, \ldots, 2^{m'}\}$, for some $m' \leq m < \log \frac{2N}{k} + 1$. Let 2^h be the size of the last bucket; then, for every bucket size other than the size of the last bucket, there are at most $\frac{k}{2} + 1$ and at least $\frac{k}{2}$ bucket of that size.*

Let $C_j = 2^r$ be the size of the j-th bucket. If Invariant 2 is satisfied, then we are guaranteed that there are at least $\frac{k}{2}$ buckets each of sizes $1, 2, 4, \ldots, 2^{r-1}$ which have indexes less than j. Consequently, $C_j \leq \frac{2}{k}(1 + \sum_{i=1}^{j-1} C_i)$. It follows that if Invariant 2 is satisfied then Invariant 1 is automatically satisfied. If we maintain Invariant 2, it is easy to see that to cover all the active 1's, we would require no more than $m \leq (\frac{k}{2} + 1)(\log(\frac{2N}{k} + 1) + 1)$ buckets. Associated with bucket is its size and a timestamp. The bucket size takes at most $\log N$ values and hence we can maintain them using $\log \log N$ bits. Since a timestamp requires $\log N$ bits, the total memory requirement of each bucket is $\log N + \log \log N$ bits. Therefore, the total memory requirement (in bits) for an EH is $O(\frac{1}{\epsilon} \log^2 N)$. It is implied that by maintaining Invariant 2, we are guaranteed the desired relative error and memory bounds.

The query time for EH is $O(1)$. We achieve this by maintaining two counters, one for the size of the last bucket (LAST) and one for the sum of the sizes of all buckets (TOTAL). The estimate itself is TOTAL minus half of LAST. Both counters can be updated in $O(1)$ time for every data element. The following is a detailed description of the update algorithm.

Algorithm Insert:

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket and update the counter LAST containing the size of the last bucket and the counter TOTAL containing the total size of the buckets.
2. If the new data element is 0 ignore it; else, create a new bucket with size 1 and the current timestamp, and increment the counter TOTAL.
3. Traverse the list of buckets in order of increasing sizes. If there are $\frac{k}{2}+2$ buckets of the same size, merge the oldest two of these buckets into a single bucket of double the size. (A merger of buckets of size 2^r may cause the number of buckets of size 2^{r+1} to exceed $\frac{k}{2}+2$, leading to a cascade of such mergers.) Update the counter LAST if the last bucket is the result of a new merger.

Example 1 We illustrate the algorithm for a few steps. Assume that $\frac{k}{2}=2$ and that the current bucket sizes from left to right are 32, 16, 8, 8, 4, 4, 2, 1, 1. When a new 1 arrives, the older 1's are merged and the bucket sizes become 32, 16, 8, 8, 4, 4, 2, 2, 1. After two more 1's arrive, the merging cascades up to the buckets of size 8, and we get buckets of sizes 32, 16, 16, 8, 4, 2, 1. ∎

Merging two buckets corresponds to creating a new bucket whose size is equal to the sum of the sizes of the two buckets and whose timestamp is the timestamp of the older bucket. A merger requires $O(1)$ time. Moreover, while cascading may require $\Theta(\log \frac{2N}{k})$ mergers upon the arrival of a single new element, standard arguments allow us to argue that the amortized cost of mergers is $O(1)$ per new data element. We obtain the following theorem:

Theorem 1 *The EH algorithm maintains a data structure which can give an estimate for the* BASICCOUNTING *problem with relative error at most ϵ using at most $(\frac{k}{2}+1)(\log(\frac{2N}{k}+1)+1)$ buckets, where $k=\lceil\frac{1}{\epsilon}\rceil$. The memory requirement is $\log N+\log\log N$ bits per bucket. The arrival of each new element can be processed in $O(1)$ amortized time and $O(\log N)$ worst-case time. At each time instant, the data structure provides a count estimate in $O(1)$ time.*

If instead of maintaining a timestamp for every bucket, we maintain a timestamp for the most recent bucket and maintain the difference between the timestamps for the successive buckets then we can reduce the total memory requirement to $O(k\log^2\frac{N}{k})$.

3 Lower Bounds

We provide a lower bound which verifies that the EH Algorithm is optimal in its memory requirement. We start with a deterministic lower bound of $\Omega(k\log^2\frac{N}{k})$.

Theorem 2 *Any deterministic algorithm that provides an estimate for the* BASICCOUNTING *problem at every time instant with relative error less than $\frac{1}{k}$ for some integer $k\leq 4\sqrt{N}$ requires at least $\frac{k}{16}\log^2\frac{N}{k}$ bits of memory.*

The proof argument will go as follows. We will show that there are a large number of arrangements of 0's and 1's such that any deterministic algorithm which provides estimates with small relative error has to differentiate between every pair of these arrangements. The number of memory bits required by such an algorithm must therefore exceed the logarithm of the number of arrangements. The above argument is formalized by the following lemma (proof in Appendix A).

Lemma 1 *For $k/4\leq B\leq N$, there exist $L=\binom{B}{k/4}^{\lfloor\log\frac{N}{B}\rfloor}$ arrangements of 0's and 1's of length N such that any deterministic algorithm for* BASICCOUNTING *with relative error less than $\frac{1}{k}$ must differentiate between any two of the arrangements.*

To prove Theorem 2, observe that if we choose $B = \sqrt{Nk}$ then $\log L \geq \frac{k}{16}\log^2\frac{N}{k}$. We also extend the lower bound on the space complexity to randomized algorithms. Proof sketches for the following two theorems can be found in Appendix C and D.

Theorem 3 *Any randomized Las Vegas algorithm for* BASICCOUNTING *with relative error less than* $\frac{1}{k}$*, for some integer* $k \leq 4\sqrt{N}$*, requires at least* $\frac{k}{16}\log^2\frac{N}{k}$ *bits of memory.*

Theorem 4 *Any randomized Monte Carlo algorithm for* BASICCOUNTING *problem with relative error less than* $\frac{1}{k}$*, for some integer* $k \leq 4\sqrt{N}$*, with probability at least* $1-\delta$ *(for* $\delta < \frac{1}{2}$*) requires at least* $\frac{k}{64}\log^2\frac{N}{k} - \log(1-\delta)$ *bits of memory.*

4 Beyond 0's and 1's

Consider now the extension of BASICCOUNTING to the case where the elements are positive integers:

Problem 2 (SUM) *Given a stream of data elements that are positive integers in the range* $[0 \ldots R]$*, maintain at every time instant the sum of the last* N *elements.*

We assume that $\log R = o(N)$. This is a realistic assumption which simplifies our calculations. We generalize EH to this setting as follows. View the arrival of a data element of value v as the arrival of v data elements with value 1 all at the same time and employ the same insertion procedure as before. Note that the algorithm in Section 2 does not require distinct timestamps, they are only required to be nondecreasing. While earlier there could be at most N active 1's, now there could be as many as NR. As before, let $k = \lceil\frac{1}{\epsilon}\rceil$. The results in in Section 2 imply that the EH will require at most $(\frac{k}{2}+1)(\log(\frac{2NR}{k}+1)+1)$ buckets. Now, each bucket will require $\log N + \log(\log N + \log R)$ bits of memory to store the timestamp and the size of the bucket. Note that there are N distinct timestamps at any point (as before), but that the bucket sizes could take on $\log N + \log R$ distinct values. Thus, the number of memory bits required is $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$. The only catch appears to be that we need $\Omega(R)$ time per insertion. The rest of the section is devoted to devising a scheme that requires only $O(\frac{\log R}{\log N})$ amortized time and $O(\log N + \log R)$ worst case time per insertion. Note that if $R = O(poly(N))$ then the amortized insertion time becomes $O(1)$ and the worst case time becomes $O(\log N)$.

Let S be the total size of the buckets at some time instant. For $j = \log(\frac{2NR}{k}+1)$, let $k_0, k_1, \ldots, k_j$ be a sequence in which k_i denotes the number of buckets of size 2^i. Then, $S = \sum_{i=0}^{j} k_i 2^i$. By Invariant 2, we have $l \leq k_i \leq l+1$, for $i < j$, and $1 \leq k_j \leq l+1$, where $l = \frac{k}{2} \geq 1$. Given $l \geq 1$ and S, a sequence $k_0, k_1, \ldots, k_j$ satisfying the above conditions is called an l-canonical representation of S. The algorithm represents every valid sum in its l-canonical form. We claim that the l-canonical representation of any sum S is unique and can be computed in time $O(\log S)$. The proof of the following Lemma can be found in Appendix B.

Lemma 2 *The* l*-canonical representation of any positive number* S *is unique.*

The following procedure computes the l-canonical representation of S in time $O(\log S)$.

Procedure l**-Canonical:** Given S find the largest j such that $2^j \leq \frac{S}{l}+1$ and let $S' = S-(2^j-1)l$. If $S' \geq 2^j$, find m such that $m2^j \leq S' < (m+1)2^j$ and set $k_j = m$; we are guaranteed that $m < l$. Let $\widehat{S} = S' - m2^j < 2^j$. Let $b_0, \ldots, b_{j-1}$ be the binary representation of $\widehat{S}$. Set $k_i = l + b_i$ for $i < j$.

Given S and l, the l-canonical representation of S tells us the exact positions of all the 1's where the buckets will start. Note that since multiple 1's "belong" to the same data element, we may

have multiple buckets starting at a single data element, implying that multiple buckets could have the same timestamp. The following observation is critical to the incremental maintenance of the buckets. The algorithm in Section 2 guarantees that if a certain data element (which in that case was some active 1) is not "indexed" at a certain time interval then it will never be "indexed" in the future. By "indexed" we mean that it is the first element of some bucket and hence its timestamp is maintained as the timestamp of that bucket. As time progresses, buckets may get merged and some data elements may not be indexed any more. However, it never happens that an element that was not indexed at some time gets indexed later.

The preceding observation allows us to devise the following scheme to incrementally maintain the buckets with small amortized update time. Let us assume that we know the buckets at a certain time instant. We think of each data element as series of 1's. We buffer B new elements separately and maintain the sum for these elements; that is, the EH is not updated for B steps. During this period, any query can be answered using a combination of the EH and the buffer sum. When the buffer gets full, we first delete any expired buckets in the EH. After the expired buckets are deleted, let S_1 be the sum of the sizes of the active buckets. Let S_2 be sum of the elements in the buffer. We calculate the l-canonical ($l = \frac{k}{2}$) representation of $S_1 + S_2$ to determine the positions of the new buckets. This requires $O(\log(S_1 + S_2)) = O(\log N + \log R)$ time since $S_1 + S_2 = O(NR)$. We then create the new buckets using the timestamps and values of the elements in the buffer, and the timestamps and sizes of the old buckets. The total time required to process the B elements in buffer is $O(B + \log N + \log R)$, since $O(B)$ time suffices to maintain the buffer sum and the number of buckets in the new histogram is $O(\log N + \log R)$. Since the time required to construct the new histogram is $O(\log N + \log R + B)$, the amortized update time per element is $O(1 + \frac{\log N + \log R}{B})$. Choosing $B = \Theta(\log N)$ makes the amortized update time $O(\frac{\log R}{\log N})$ and worst case time $O(\log N + \log R)$. The buffer needs $O(\log N(\log N + \log R))$ memory bits, which is the same as the memory requirement of the EH. Note that if R is $poly(N)$ then the amortized update time is $O(1)$ and worst case time is $O(\log N)$. We have obtained a memory upper bound of $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ bits, as summarized in the following theorem.

Theorem 5 *The generalized EH for the* SUM *problem maintains a data structure which provides estimates with relative error at most ϵ using at most $(\frac{k}{2} + 1)(\log(\frac{2NR}{k} + 1) + 1)$ buckets, where $k = \lceil \frac{1}{\epsilon} \rceil$. The memory requirement is $\log N + \log(\log N + \log R)$ bits per bucket. The arrival of each new element can be processed in $O(\frac{\log R}{\log N})$ amortized time and $O(\log N + \log R)$ worst case time. At each time instant, the data structure provides a sum estimate in $O(1)$ time.*

We now prove a lower bound of $\Omega(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ bits. If $\log N = \Omega(\log R)$ then the lower bound from Section 3 applies. Thus, we only need to consider the case when $R > N$. We will assume that $\log R \leq \frac{N}{k}$; in fact we assume $\log R = o(N)$. Consider the following arrangements. We break the window of size N into $\log R$ blocks, each of size $\lfloor \frac{N}{\log R} \rfloor$. Consider the i-th block, for $0 \leq i < \log R$. We choose $k/4$ of the $\lfloor \frac{N}{\log R} \rfloor$ positions and place an element with value 2^i there, setting all other elements to 0. By an argument similar to the one in Section 3, any deterministic algorithm with relative error less than $\frac{1}{k}$ must differentiate between any two of these arrangements. The total number of these arrangements is $\binom{N/\log R}{k/4}^{\log R} \geq (\frac{4N}{k \log R})^{\frac{k}{4} \log R}$. The number of memory bits required is at least $\frac{k}{4} \log R \log(\frac{4N}{k \log R}) = \Omega(\frac{1}{\epsilon}(\log N + \log R)(\log N))$. We assume that $R > N$ and that $\log R = O(N^\delta)$ for some $\delta < 1$. Note that the lower bounds also apply for randomized algorithms that provide an approximate answer.

5 Computing L_p norms for vectors

We now extend the EH technique and combine it with the sketching technique from Indyk [11] to compute the L_p norms of vectors in the sliding window model. Assume that the window is broken into smaller contiguous buckets. These are numbered right to left and are denoted by $B_1, B_2, \ldots, B_m$. Consider a function f, defined over the intervals, with the following properties:

P1: $f(B_i) \geq 0$.

P2: $f(B_i) \leq poly(|B_i|)$.

P3: $f(B_1 + B_2) \geq f(B_1) + f(B_2)$, with $B_1 + B_2$ the concatenation of adjacent buckets B_1 and B_2.

P4: $f(B_1 + B_2) \leq C_f(f(B_1) + f(B_2))$, where $C_f \geq 1$ is a constant.

P5: The function $f(B)$ admits a "sketch" which requires $g_f(|B|)$ space and is composable, i.e., the sketch for $f(B_1 + B_2)$ can be composed efficiently from the sketches for $f(B_1)$ and $f(B_2)$.

If the function f admits these properties then we can efficiently estimate it for sliding windows using the EH technique. We maintain buckets with the following two invariants; we also associate with every bucket a timestamp and the sketch.

Invariant 3 $f(B_{n+1}) \leq \frac{C_f}{k} \sum_{i=1}^{n} f(B_i)$.

Invariant 4 $f(B_{n+2}) + f(B_{n+1}) > \frac{1}{k} \sum_{i=1}^{n} f(B_i)$.

Observation 1: We estimate the function f for the current window by composing the sketches of all but the earliest (leftmost) bucket. The leftmost bucket may have certain expired data elements along with a suffix of data elements that are active. Let B_x be the part (suffix) of the leftmost bucket that is active and was ignored (did not contribute to the estimate). Let B_y be the concatenation of the all the other buckets whose sketch we compose using the sketches of the individual buckets. Then $B_x + B_y$ is the current window and the exact answer is $f(B_x + B_y)$. However we estimate the answer as $f(B_y)$; thus, we always underestimate. The relative error E_r is $\frac{f(B_x+B_y)-f(B_y)}{f(B_x+B_y)} > 0$, by **P1** and **P3**. Also we have

$$\begin{aligned}
E_r &\leq \frac{f(B_x+B_y)-f(B_y)}{f(B_y)} && \textbf{(P1, P3)} \\
&\leq \frac{C_f(f(B_x)+f(B_y))-f(B_y)}{f(B_y)} && \textbf{(P4)} \\
&= \frac{C_f f(B_x)}{f(B_y)} + C_f - 1 \\
&\leq \frac{C_f f(B_{n+1})}{\sum_{i=1}^{n} f(B_i)} + C_f - 1 && \textbf{(P1, P3)} \\
&\leq \frac{C_f^2}{k} + C_f - 1 && \text{(Invariant 3)}
\end{aligned}$$

Observation 2: Invariant 4 and property **P2** imply that the number of buckets will be $O(k \log N)$, where N is the size of the window. Thus, the memory required to maintain the time-stamp and sketches for all the buckets will be $O(k \log N(\log N + g_f(N)))$.

Hence, if we maintain the invariants along with the timestamp and the sketches, we can estimate the function f with relative error $0 \leq E_r \leq \frac{C_f^2}{k} + C_f - 1$ using $O(k \log N(\log N + g_f(N)))$ memory bits. We can maintain the invariants along with the timestamp and sketches as new data elements are added. The algorithm to do this is very similar to that for EH.

1. When a new data element arrives, calculate the new expiry time. If the timestamp of the last bucket indicates expiry, delete that bucket.
2. Create a new bucket with just the new data element.
3. Traverse the list of buckets from right to left. If Invariant 4 is violated for a pair of buckets (B_{n+1}, B_{n+2}), merge them into a new bucket B'_{n+1}. The sketch for this bucket is composed from the sketches for B_{n+1} and B_{n+2}. We may need to do more than one merge.

We argue that the algorithm maintains Invariant 3 and Invariant 4. Adding a new bucket does not violate Invariant 3, as we only increase the size of the suffix. Whenever Invariant 4 is violated, the two buckets involved satisfy $(f(B_{n+2}) + f(B_{n+1}) \leq \frac{1}{k}\sum_{i=1}^{n} f(B_i))$. When we merge them, property **P4** guarantees that $f(B'_{n+1}) \leq \frac{C_f}{k}\sum_{i=1}^{n} f(B_i)$ and hence Invariant 3 is valid for the new bucket B'_{n+1}. The algorithm may need to do a lot of merges, as many as the number of buckets $(O(\log N)))$. However, the amortized time is $O(1)$. We omit details dealing with the fact that the function f for a window of size 1 may be greater than 1 although bounded by some constant R.

Theorem 6 *A function f with properties* **P1–P5** *can be estimated over sliding windows with relative error $0 \leq E_r \leq \frac{C_f^2}{k} + C_f - 1$ using $O(k \log N(\log N + g_f(N)))$ bits of memory.*

L_p Norms We now argue that L_p norms (for $p \in [1, 2]$) of vectors under a restricted model admit the properties **P1–P5** and hence can be efficiently computed for sliding windows. Consider the restricted model [11] in which each data element is a pair (i, a), where $i \in [d] = 0 \ldots d-1$ and $a \in 0 \ldots M$ represents an increment to the ith dimension of an underlying vector. Every window B represents a vector and its $L_p(B)$ norm is given by $L_p(B) = (\sum_{i \in [d]} |\sum_{(i,a) \in B} a|^p)^{1/p}$.

Note that the case $p = 1$ is the same as the SUM problem. We denote $(L_p)^p$ by f_p and estimate f_p, for $p \in [1, 2]$. The function f_p clearly admits properties **P1–P4**. For **P5**, $f_p(B)$ admits a sketching technique which requires $O(\log M \log(1/\delta)/\hat{\epsilon}^2)$ memory bits per sketch and is composable. The technique also requires $O(\log M \log(d/\delta) \log(1/\delta)/\hat{\epsilon}^2)$ random bits which are common to all sketches. (See Theorem 2 in [11].) The sketches for computing the function are not exact. They provide an approximation with relative error less than $\hat{\epsilon}$ with probability δ. However, by setting the accuracy parameter $\hat{\epsilon}$ correctly we can make sure that our algorithm also works in an probabilistic manner and has relative error at most $\frac{4}{k} + 1 + \epsilon$, where ϵ is a function of $\hat{\epsilon}$.

This proves that under the restricted model we can compute $(L_p)^p$ with relative error at most $\frac{4}{k} + 1 + \epsilon$ using $O(k \log N(\log N + \log M \log(1/\delta)/\hat{\epsilon}^2))$ bits of memory. The estimate is probabilistically approximately correct. Note that computing $(L_p)^p$ with small relative error translates to computing L_p with a small relative error.

Lower Bounds The BASICCOUNTING and SUM problems are the special cases of computing L_p norms, where the underlying vector has a single dimension. Thus, the lower bounds for these problems apply to the problem of computing the L_p norm. Note that the upper bounds obtained in this section match the lower bounds. The L_p norm for $p = 0$ is defined as the distinct value problem and we deal with this problem in Section 6.

6 Applications

We briefly discuss how the EH algorithm for BASICCOUNTING can be used as a building block to adapt several techniques to the sliding window model with a multiplicative overhead of $O(\frac{1}{\epsilon}\log N)$ in memory and a $1 + \epsilon$ factor loss in accuracy. The basic idea is that to adapt to the sliding window

setting a scheme relying on exact counters for positive integers, use an EH to play the role of a counter. A counter would have required $\Omega(\log N)$ memory bits, while EH requires $O(\frac{1}{\epsilon}\log^2 N)$ memory bits and maintains the count within $1+\epsilon$ error.

Hash Tables: This is the simplest case. Every data element gets hashed to a bucket. Instead of maintaining a counter for each bucket, we use the EH to maintain approximate counts of the number of data elements hashed into the bucket from the last N data elements in the stream. This works if we are required to maintain only the count of elements in each hash bucket.

Sums and Averages: In Section 4, we showed to maintain the sum of positive integer data elements using the generalized version of the EH. This requires $O(\frac{1}{\epsilon}\log N(\log R + \log N))$ bits of memory. Since maintaining sum would require $\log N + \log R$ bits the multiplicative overhead is $O(\frac{1}{\epsilon}\log N)$. Maintaining averages is similar.

Histograms: Given the bucket boundaries in a histogram, we can maintain the sum, average, and other statistics corresponding to each bucket using the generalized EH. Finding the optimal bucket boundaries to optimize the memory requirement is an orthogonal problem. Also equiwidth histograms are a natural choice of histograms for which the bucket boundaries are fixed. Note that unlike the histograms discussed in [8] these are not time-based histograms, but instead could be based on any attribute of the data.

Min and Max: We prove a lower bound for the memory requirement of an algorithm that maintains min or max over a sliding window. The lower bound is based on a counting argument like the one used to prove the lower bound for BASICCOUNTING. Let the data elements be drawn from a set of R distinct numbers. Consider all *nondecreasing* arrangements of N numbers. The number of such arrangements is $\binom{N+R}{N}$. Any deterministic algorithm that gives the correct answer at every time instant must differentiate between any two such arrangements. This is because the two arrangements will have a different minimum at the first place that they differ from left to right. The lower bound on the number of memory bits required is then $\log\binom{N+R}{N} \geq N\log(R/N)$. This lower bound is also valid for any randomized algorithms by arguments similar to the one in Section 3. If $R = poly(N)$ then the lower bound says that we have to store all of the last N elements. The easiest way to maintain the exact minimum over sliding windows is to maintain a list of pairs (value, timestamp) such that both the value and the timestamp are strictly increasing. This scheme has a worst-case space requirement of $O(N\log R)$ bits. However, if the data elements arrive in a random order, the expected space complexity will be $O(\log N \log R)$.

Distinct Values: It is easy to adapt the technique of Flajolet and Martin [6] to estimate the number of distinct elements in the last N data elements. Their *probabilistic counting technique* maintains a bitmap of size $O(\log R)$, where R is an upper bound on the number of distinct values in the data set. In the case of sliding windows, $R \leq N$ and a bitmap of size $O(\log N)$ suffices. We also maintain with each bit a timestamp of size $O(\log N)$. Whenever a bit is (re)set by a data element we update the timestamp to that of the data element. This enables us to keep track of the bits that were set by the last N elements. Consequently, we can estimate the number of distinct elements with an expected relative accuracy of $O(\frac{1}{\sqrt{m}})$ using $O(m\log^2 N)$ bits of memory. Note that the lower bound for BASICCOUNTING problem applies to the Distinct Value problem. Given an instance of BASICCOUNTING problem we can create an input where a 0 is mapped to 0 while every 1 is mapped to some distinct value (the *arrival time* of the element for instance). Then the number of Distinct Values is one more than the number of ones. This reduction shows that the lower bounds for BASICCOUNTING problem apply to the Distinct Value problem.

References

[1] N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 1996.

[2] C. Cortes, K. Fisher, D. Pregibon, A. Rogers. Hancock: a language for extracting signatures from data streams. In *Proc. 2000 ACM SIGKDD*, pp. 9–17, 2000.

[3] S. Chaudhuri, R. Motwani, V. R. Narasayya. On random sampling over joins. In *Proc. 1999 ACM SIGMOD*, pp. 263–274, 1999.

[4] M. Fang, H. Garcia-Molina, R. Motwani, N. Shivakumar, J.D. Ullman. Computing iceberg queries efficiently. In *Proc. 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[5] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. In *Proc. 40th Symposium on Foundations of Computer Science*, 1999.

[6] P. Flajolet, G. Martin. Probabilistic Counting. In *Proc. 24th Symposium on Foundations of Computer Science*, 1983.

[7] C. Fraleigh, S. Moon, C. Diot, B. Lyles, F. Tobagi. Architecture of a passive monitoring system for backbone IP networks. Technical Report TR00-ATL-101-801, Sprint Labs, 2000.

[8] S. Guha, N. Koudas, K. Shim. Data-Streams and Histograms. To appear in *Proc. Thirty-Third Annual ACM Symposium on Theory of Computing*, 2001.

[9] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan. Clustering data streams. In *Proc. 2000 Annual IEEE Symp. on Foundations of Computer Science*, pages 359–366, 2000.

[10] M. R. Henzinger, P. Raghavan, S. Rajagopalan. Computing on data streams. Technical Report TR 1998-011, Compaq Systems Research Center, Palo Alto, California, May 1998.

[11] P. Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation. In *Proc. 41st Symposium on Foundations of Computer Science*, 2000.

[12] Jagadish, Koudas, Muthukrishnan, Poosala, Sevcik, Suel. Optimal Histograms with Quality Guarantees. In *Proc. 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[13] R. Motwani, P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[14] *Netflow Services and Applications.* Whitepaper, Cisco Systems, 2000. Available at http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm.

Appendix

A Proof of Lemma 1 in Section 3

Lemma 1 *For $k/4 \leq B \leq N$, there exist $L = \binom{B}{k/4}^{\lfloor \log \frac{N}{B} \rfloor}$ arrangements of 0's and 1's of length N such that any deterministic algorithm for* BASICCOUNTING *with relative error less than $\frac{1}{k}$ must differentiate between any two of the arrangements.*

Proof: We partition a window of size N into blocks of size $B, 2B, 4B, \ldots, 2^jB$ from right to left, for $(j = \lfloor \log \frac{N}{B} \rfloor - 1)$. Consider the i-th block of size 2^iB and subdivide it into B contiguous subblocks of size 2^i. For each block, we choose $\frac{k}{4}$ subblocks and populate them with 1's, placing 0's in the remaining positions. In every block, there are $\binom{B}{k/4}$ possible ways to place the 1's, and therefore the total number of distinct arrangements is $L = \binom{B}{k/4}^{\lfloor \log N/B \rfloor}$.

We now argue that any deterministic algorithm for BASICCOUNTING with relative error less than $\frac{1}{k}$ must differentiate between any pair of these arrangements. In other words, if there exists a pair of

arrangements A_x, A_y such that a deterministic algorithm does not differentiate between them, then after some time interval the two arrangements will have different answers to the BASICCOUNTING problem and the algorithm will give a relative error of at least $\frac{1}{k}$ for one of them. To this end, we will assume that the algorithm is presented with one of these L arrangements of length N, followed by a sequence of all 0's of length N.

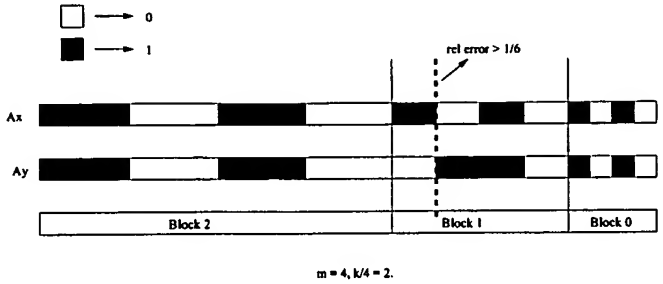


Figure 2: A pair of arrangements that should be differentiated by any deterministic algorithm with relative error less than 1/8.

Refer to Figure 2 for an illustration of a pair of arrangements that should be differentiated by any deterministic algorithm with relative error less than $\frac{1}{8}$.

Consider an algorithm that does not differentiate between two of the above arrangements A_x and A_y. We will use the numerical sequences $x_0, x_1, \ldots, x_j$ and $y_0, y_1, \ldots, y_j$, for $j = \lfloor \log \frac{N}{B} \rfloor - 1$, to encode the two arrangements. The i-th number in the sequence specifies the choice of the $k/4$ subblocks from the i-th block which are populated with 1's. The two sequences must be distinct since the two arrangements being encoded are distinct. Let d be an index of a point where the two sequences differ, i.e., $x_d \neq y_d$. Then the two arrangements have a different choice of $k/4$ subblocks in the d-th block. Number the subblocks within block d from right to left, and let h be the highest numbered subblock that is chosen for one of the arrangements (say A_x) but not for the other (A_y).Consider the time instant when this subblock h expires. At that instant, the number of active subblocks in block d for arrangement A_x is c, where $c + 1 \leq k/4$, while the number of active subblocks in block d for A_y is $c + 1$. Since the arrangements are followed by a sequence of 0's, at this time the correct answer for A_x is $c2^d + \frac{k}{4}(2^d - 1)$, while for A_y the correct answer is $(c+1)2^d + \frac{k}{4}(2^d - 1)$. Thus, the algorithm will give an absolute error of at least 2^{d-1} for one of the arrangements, which translates to a relative error of $\frac{1}{k}$ at that point in time. ■

B Proof of Lemma 2 in Section 4

Lemma 2 *The l-canonical representation of any positive number S is unique.*

Proof: We give a proof by contradiction. Assume that $\mathbf{k} = (k_0, k_1, \ldots, k_j)$ and $\mathbf{k'} = (k'_0, k'_1, \ldots, k'_{j'})$ are two distinct l-canonical representations of S. Without loss of generality, assume that $j \leq j'$. Let d be the smallest index where the sequences differ. We have $d \leq j$ since it cannot happen that they agree on all the indices less than or equal to j and the second sequence has nonzero components for indices greater than j, given that they have the same sum.

Case1 ($d < j$): Since $l \leq k_d, k'_d \leq l + 1$, we have $|\sum_{i=0}^{d} k_i 2^i - \sum_{i=0}^{d} k'_i 2^i| = 2^d$. However, $|\sum_{i=d+1}^{j} k_i 2^i - \sum_{i=d+1}^{j'} k'_i 2^i| = c2^{d+1}$, for some $c \geq 0$, which is a contradiction since $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j'} k'_i 2^i| = 0$.

Case2 ($d = j$): The sequence $\mathbf{k}'$ must have nonzero indices greater than j, otherwise the two representations cannot give the same sum. Moreover, it cannot happen that $k_j \leq k'_j$, since otherwise $\mathbf{k}'$ will have a strictly greater sum. Thus, $k_j > k'_j$ and $k_j \leq l + 1$. Since k'_j is not the last index, we have $k'_j \geq l$. Therefore $|k'_j - k_j| \leq 1$, which implies $|\sum_{i=0}^{j} k_i 2^i - \sum_{i=0}^{j} k'_i 2^i| \leq 2^j$. However, $\sum_{i \geq j+1}^{k'_i} 2^i \geq 2^{j+1}$ which gives a contradiction. ■

C Proof of Theorem 3 in Section 3

Theorem 3 *Any randomized Las Vegas algorithm for* BASICCOUNTING *with relative error less than* $\frac{1}{k}$*, for some integer* $k \leq 4\sqrt{N}$*, requires at least* $\frac{k}{16} \log^2 \frac{N}{k}$ *bits of memory.*

Proof Sketch: Define an algorithm A to be ϵ-correct for an input instance I if the value returned by A on input I has relative error less than ϵ. Yao Minimax Principle [13] implies that the expected space complexity of the optimal ϵ-correct deterministic algorithm for an arbitrarily chosen input distribution $\mathbf{p}$ is a lower bound on the expected space complexity of the optimal ϵ-correct Las Vegas randomized algorithm. Consider the uniform distribution over the input arrangements in Lemma 1. Then any deterministic algorithm that is ϵ-correct for all these instances must differentiate between any two distinct arrangement. As a result the expected space complexity of an optimal deterministic algorithm on this distribution is at least equal to the optimal coding length for the probability distribution. Since the coding length is at least equal to the entropy of the distribution, we get the same lower bound (logarithm of the number of instances) as in the case of a deterministic algorithm. This proves the generalization of Theorem 2 to Las Vegas randomized algorithms. ■

D Proof of Theorem 4 in Section 3

Theorem 4 *Any randomized Monte Carlo algorithm that for* BASICCOUNTING *problem with relative error less than* $\frac{1}{k}$*, for some integer* $k \leq 4\sqrt{N}$*, with probability at least* $1 - \delta$ *(*$\delta < \frac{1}{2}$*)requires at least* $\frac{k}{64} \log^2 \frac{N}{k} - \log(1 - \delta)$ *bits of memory.*

Proof Sketch: We use the analogous version of Yao's Minimax Principle for Monte Carlo randomized algorithms [13] to establish lower bound for Monte Carlo algorithms. Consider a deterministic algorithm that is ϵ-correct with probability at least $1 - \delta$, for some $\delta < \frac{1}{2}$. As before the input distribution $\mathbf{p}$ that we consider is the uniform distribution over all the arrangements defined in Lemma 1. Since the deterministic algorithm is ϵ-correct with probability at least $1 - \delta$, it is ϵ-correct for at least $1 - \delta$ fraction of the inputs. Thus by arguments similar to those in the previous section of the Appendix we get the same lower bound except for an additive loss of $\log(1 - \delta)$ and a multiplicative loss of $\frac{1}{4}$. Asymptotically, the lower bound does not change. ■

Streaming Queries over Streaming Data

Sirish Chandrasekaran Michael J. Franklin

University of California at Berkeley
{sirish,franklin}@cs.berkeley.edu

Abstract

Recent work on querying data streams has focused on systems where newly arriving data is processed and continuously streamed to the user in real-time. In many emerging applications, however, ad hoc queries and/or intermittent connectivity also require the processing of data that arrives prior to query submission or during a period of disconnection. For such applications, we have developed PSoup, a system that combines the processing of ad-hoc and continuous queries by treating data and queries symmetrically, allowing new queries to be applied to old data and new data to be applied to old queries. PSoup also supports intermittent connectivity by separating the computation of query results from the delivery of those results. PSoup builds on adaptive query processing techniques developed in the Telegraph project at UC Berkeley. In this paper, we describe PSoup and present experiments that demonstrate the effectiveness of our approach.

1 Introduction

The proliferation of the Internet, the Web, and sensor networks have fueled the development of applications that treat data as a continuous stream, rather than as a fixed set. Telephone call records, stock and sports tickers, and data feeds from sensors are examples of streaming data. Recently, a number of systems have been proposed to address the mismatch between traditional database technology and the needs of query processing over streaming data (e.g., [HFCD+00, AF00, CDTW00, BW01, CCCC+02]).

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599, and by IBM, Microsoft, Siemens, and the UC MICRO program.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

In contrast to traditional DBMSs, which answer streams of queries over a non-streaming database, these *continuous query (CQ)* systems treat queries as fixed entities and stream the data over them.

Previous systems allow only the queries or the data to be streamed, but not both. As a result, they cannot support queries that require access to both data that arrived previously and data that will arrive in the future. Furthermore, existing CQ systems continuously deliver results as they are computed. In many situations, however, such continuous delivery may be infeasible or inefficient. Two such scenarios are:

Data Recharging: Data Recharging [CFZ01] is a process through which personal devices such as PDAs periodically connect to the network to refresh their data contents. For example, consider a business traveler who wishes to stay apprised of information ranging from the movements of financial markets to the latest football scores, all within a certain historical window. These interests are encoded into queries to be executed at a remote server, the results of which must be downloaded to the user's PDA when it is connected to the network infrastructure.

Monitoring: Consider a user who wants to track interesting pieces of information such as the number of music downloads from within his subnet in the last hour, or recent postings on Slashdot (http://www.slashdot.org/) with a *score* greater than a certain threshold. Even when online, the user might only periodically wish to see summaries of recent activity, rather than being interrupted by every update. Aggregated over many users, the bandwidth and server load wasted on transmitting data that is never accessed will be significant. A more efficient approach is to return the current results of a standing query *on demand.*

To support such applications, we propose PSoup, a query processor based on the Telegraph [HFCD+00] query processing framework. The core insight in PSoup that allows us to support such applications is that both data and queries are streaming, and more importantly, they are duals of each other: *multiquery processing is viewed as a join of query and data streams*. In addition, PSoup also partially materializes results to support disconnected operation, and to improve data throughput and query response times.

1.1 Overview of the System

A user interacts with PSoup by initially *registering* a query specification with the system. The system returns a handle

to the user, which can then be used repeatedly to *invoke* the results of the query at later times. A user can also explicitly unregister a previously specified query.

An example query specification is shown below:

```
SELECT *
FROM Data_Stream D_S
WHERE (D_s.a < v1 ∧ D_s.b > v2)
BEGIN (NOW − 10)
END (NOW),
```

PSoup supports SELECT-FROM-WHERE queries with conjunctive predicates.[1] Queries also contain a BEGIN-END clause that specifies the input window over which the query results are to be computed. In this paper, we assume that the system clock time is used to define the ends of the input window, and that the same time-window applies to all the streams in the FROM clause. The ideas presented here can be adapted to allow logical windows (i.e., based on the number of tuples, rather than system clock time), and the application of different windows for each stream. The arguments to the BEGIN-END clause can either be constants (using absolute values), or can be specified relative to the current system clock (using the keyword NOW). The BEGIN-END clause allows the specification of *snapshot* (constant BEGIN_TIME, constant END_TIME) [SWCD97], *landmark* (constant BEGIN_TIME, variable END_TIME) or *sliding window* (variable BEGIN_TIME, variable END_TIME) semantics [GKS01] for the queries. Because PSoup is currently implemented as a main-memory engine, the acceptable windows are limited by the size of memory.

Internally, PSoup views the execution of a stream of queries over a stream of data as a join of the two streams, as illustrated in Figure 1. We refer to this process as the query-data join.

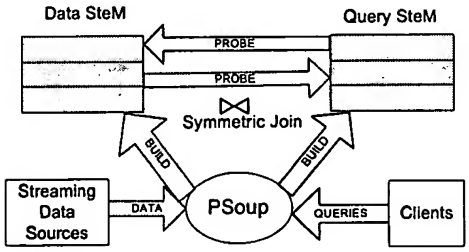


Figure 1: Outline of solution

Our system maintains structures called *State Modules (SteMs)* [Ram01] for the queries and the data. There is one Query SteM for all the query specifications in the system, and there is one Data SteM for each data stream. Figure 1 shows an example with one data stream. When a client first registers a query, it is inserted into the Query SteM, and then used to probe the Data SteM. This application of "new" queries to "old" data is how PSoup executes queries over historical data. Similarly, when a new data element arrives, it is inserted into the Data SteM, and used to probe the Query SteM. This act of applying "new" data to "old" queries is how PSoup supports continuous queries. In both cases, the results of the probes are materialized in a Results Structure (not shown in figure). When a query is invoked, the current input window is computed from the BEGIN-END clause using the current value of *NOW*. This window is then applied to the materialized values to retrieve the current results. Materialization is the key to efficient support for set-based semantics in continuous queries.

[1] The system currently does not allow nested subqueries. This constraint is not inherent in the treatment of queries as data. The implementation of subqueries is the subject of future work.

1.2 Contributions of the paper

We propose a scheme that efficiently solves the problem of intermittently repeated snapshot, landmark and sliding window queries over streaming data within a recent historical window. We explore the tradeoff between the computation required to materialize and maintain the results of a query and the response time for invocation of those queries.

We demonstrate several advantages of treating data and queries as streams, and as duals. First, this idea is the key to solving the problem of processing queries that can access both data that arrived before the query registration, and also data that will arrive in the future. Second, multiquery evaluation can be optimized by using appropriate algorithms to join the data and query streams. Third, we can leverage eddies[AH00] to adaptively respond to changing characteristics of both the data and query specification streams.

Finally, we develop techniques to share both the computation and storage of different query results. We index predicates to share computation for incremental maintenance across standing queries. The storage of the results of the query-data join computation is the key to PSoup's ability to support intermittently connected operation. We share storage across the base data and the results of all standing queries by avoiding copies.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Sections 3 and 4 we describe how PSoup executes Selection and Join Queries. We present the results of our experiments in Section 5. Section 6 discusses issues involving Aggregation queries that are of specific interest to PSoup. In Section 7 we present our conclusions and directions for future work.

2 Related Work

PSoup is part of the Telegraph[HFCD+00] project at UC Berkeley. It spans work on continuous queries, triggers and materialized views.

Terry et al. [TGNO92] studied continuous queries to filter documents using a SQL-like language that only allows monotonic queries. Seshadri et al.[SLR94] discuss the problem of defining and executing database-style queries over sequenced data. They only consider queries that produce a singleton tuple as output for each input window. Sadri et al. [SZZA01] propose a language SQL-TS, that can express sequence-sensitive operations over windows of the stream. A key feature of SQL-TS is the ability to define windows according to repeating patterns in the stream.

Recently, various CQ engines have been proposed in the literature. PSoup builds on the ideas developed in CACQ [MSHR02], an earlier CQ extension of the Tele-

graph engine that exploits the adaptivity offered by the Eddy operator [AH00] to efficiently handle skews in data distribution and arrival rates. CACQ also introduced the notion of *tuple-lineage* to allow sharing queries beyond just common subtrees of the plans. Other systems [YG99, CDTW00, AF00] have explored less adaptive techniques to support continuous queries. All these four systems focus on "filter" operators: they accept one long sequence of tuples as input and produce another monotonically growing sequence as output. Further, they do not offer support for queries over historical data. Compared to these systems, we consider a more comprehensive workload, allowing queries to have non-monotonic sets as inputs and output, thereby allowing *snapshot*, *landmark* and *sliding* window queries. The techniques developed in PSoup to query recently arrived and future data, and to support disconnected operation can be integrated into these earlier CQ systems. In some ways, PSoup can be seen as a logical extension of these CQ techniques to handle intermittent set-based queries over both recent and future data.

Fabret et al. [FJLP+01] observe that publish-subscribe systems can apply newly published events to existing subscriptions, and match new subscriptions to existing (valid) events. However, they focus on grouping subscriptions and optimizing the matching process on the arrival of new data. and suggest that standard query processing techniques can be used to process new subscriptions.

Bonnet et al.[BGS01, BS00] describe different kinds of queries over streaming data. Fjords[MF02] is an architecture for querying streaming sensor data. MOST[SWCD97] is a database for querying moving objects, and considers semantic issues for time-based specification of queries. STREAM[BW01] considers the relation of materialized views to continuous queries in the context of self-maintenance. In our work, we are less concerned with the tradeoff between computation and scratch storage, than with the sharing of storage among different queries.

Other recent research[LSM99, GKS01, SH98] has focused on developing algorithms to perform specific functions on sequenced data. Instead, we focus on general Select-Project-Join (SPJ) views and simple classes of aggregates.

The computation of standing queries based on tuple-windows is similar to trigger processing and the incremental maintenance of materialized views. Triggerman [HCHK+99] is a scalable trigger system that uses the Gator discrimination network [HBC97] to statically compute optimal strategies for processing the trigger. Gator is a generalization of the Rete [F82] and TREAT [M87] algorithms. The Chronicle data model [JMS95] defines an algebra for the materialized view problem over append-only data. Wave indices [SG97] are another solution designed for append-only data in a data warehousing scenario. They are a set of indices maintained over different time-intervals of the data, and allow queries over windowed input. They ensure high *harvest* [FGCB+97] (i.e., fraction of data used to answer query) of the data while old data is being expired, or as new data arrives. This technique works well for hourly or daily bulk data updates but does not scale to higher data arrival and expiration rates.

3 Query Processing Techniques

In this section we describe how PSoup processes a stream of queries having the same FROM clause using several examples. In Section 4, we extend the solution to handle queries with different FROM clauses and describe the implementation in more detail.

3.1 Overview

As described in Section 1.1, the client begins by registering a query specification with the system. Query specifications are of the form:

```
SELECT select_list
FROM from_list
WHERE conjoined_boolean_factors
BEGIN begin_time
END end_time
```

PSoup assigns the query a unique ID (called queryID) that it returns to the user as a handle for future invocations. The client can then go away (or disconnect), and return intermittently to invoke the query to retrieve the current results. Between the invocations of the query by the client, PSoup continuously matches data to query predicates in the background and materializes the results of the matches in the Results Structure. Upon invocation of the query, PSoup computes the current input window for the query using the BEGIN-END clause and applies it to the Results Structure to return the current results of the query.

3.2 Entry of new query specifications or new data

We now describe the background query-data join processing in greater detail. We defer the discussion of query invocation and of the Results Structure until Section 3.3.

When PSoup receives a query specification, it splits the query specification into two parts. The first part consists of the SELECT-FROM-WHERE clauses of the specification, which we refer to as a *standing query clause (SQC)*. The second part, which consists of the BEGIN-END clause, is stored in a separate structure called the WindowsTable for reference during future invocations of the query. The SQC is first inserted into a data structure called the Query SteM. The SQC is then used to probe the Data SteMs corresponding to the tables in its FROM clause. The Data SteMs contain the data tuples in the system. The results of the probe indicate the data tuples that satisfied the SQC. The identities of those tuples are stored in the Results Structure.

When a new data tuple enters PSoup, it is assigned a globally unique tupleID and a physical timestamp (called its physicalID) corresponding to the system clock. Next, the data tuple is inserted into the appropriate Data SteM (there is one Data SteM for each stream). The data tuple is then used to probe the Query SteM to determine which SQCs it satisfies. As we will describe in Section 3.2.2, the data tuple might be used to further probe other Data SteMs to evaluate Join queries. As before, the tupleIDs and physicalIDs of the results of the probe are stored in the Results Structure.

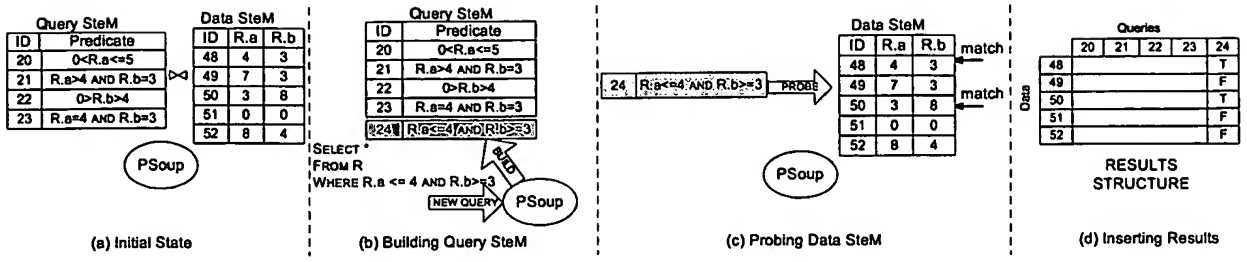


Figure 2: Selection Query Processing: Entry of New Query

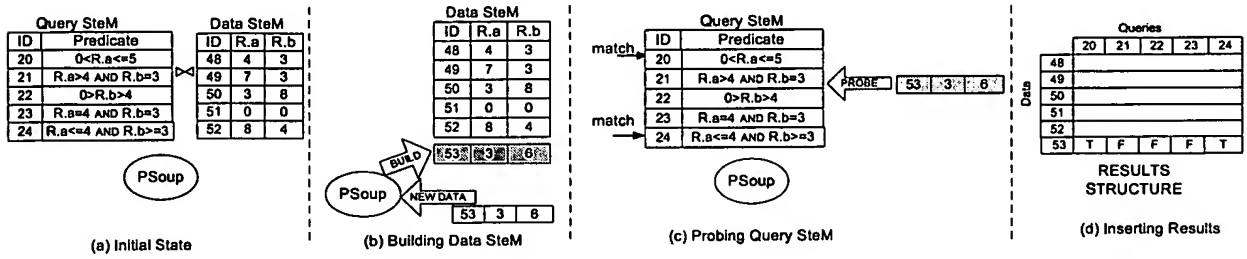


Figure 3: Selection Query Processing: Entry of New Data

We now describe this process in more detail using several examples.

3.2.1 Selection Queries over a single stream

We begin by considering simple queries that involve only a single data stream. Figure 2 illustrates the actions performed by PSoup when a new SQC enters the system. Figure 2(a) shows the state of the Query SteM and Data SteM after the system has processed the queries with queryIDs up to and including 23, and the data tuples with tupleIDs up to and including 52. Now, consider the entry of a new SQC into the system shown in Figure 2(b) (we omit the BEGIN-END clause in the figure). This standing query is assigned the queryID 24, and is inserted into the Query SteM by adding a (*queryID, QueryPredicate*) entry to the SteM. At this time, we also have to augment the Results Structure (Figure 2(d)) with a new column to store the results of the query. This standing query is then sent to probe the Data SteM, where it is matched with each data tuple (Figure 2(c)). When tuples are found to satisfy a query (data tuples with tupleIDs 48 and 50 in the figure), the appropriate entries in the Results Structure are marked TRUE (Figure 2(d)).

Analogously (as shown in Figure 3), when a new data tuple arrives it is first added to the Data SteM, and then sent to the Query SteM, where it is matched with all of the standing queries in the system. Lastly, the Results Structure is updated.

3.2.2 Join Queries over Multiple Streams

For queries over multiple data streams (i.e., Join queries), we use the same approach as before and treat the processing of multiple Join queries as a join of the query stream with all the data streams enumerated in the FROM-list of the queries. To do this, we generalize the symmetric join to accept more than two input streams.

Again, we demonstrate our solution using an example. For simplicity, we consider queries over two data streams R and S. Figure 4 shows the actions performed in PSoup when a new query enters the system. The system has already processed R and S data tuples with tupleIDs up to and including 54, and queries with IDs up to and including 22. There are two Data SteMs, one for each data stream. There is only a single Query SteM for the query stream. The SteMs have been populated with the above data and queries.

Consider the arrival of a new standing query with ID 23 (Step 1). Its predicate has factors involving only R (R.a < 5), only S (S.b > 1), and both (R.a > S.b). The query is first inserted into the Query SteM (Step 2). Next, the query is used to probe either the R or S Data SteM. Without loss of generality, let us assume that the query first probes the R Data SteM. We match each tuple in the Data SteM to this query tuple (Step 3). Because the query depends also on S, it cannot be fully evaluated at this stage. However, the R-only boolean factors can still completely evaluated to filter out those R tuples that cannot be in the final result. For the tuples that satisfy the R-only boolean factors of the query, the values for R are substituted in the join boolean factors that relate the two streams; after the substitution, there remain a set of boolean factors that depends solely on S. Next, we output a "hybrid struct" that has for each matching R tuple, the contents of the R tuple augmented with the partially evaluated predicate of the query (Step 4). Each of the hybrid structs that are thus produced are then used to probe the S Data SteM (Step 5). Here, for each S tuple that satisfies the remaining boolean factors of the query, the Results Structure is updated as follows: an entry for the pair (R-tupleID, S-tupleID) is created and inserted in the Results Structure for this pair if one does not already exist.

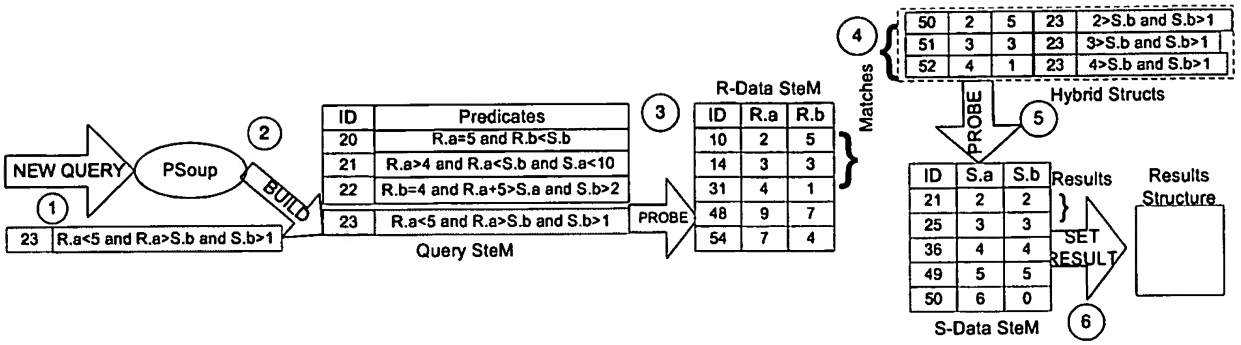


Figure 4: Join Processing: Entry of New Join Query

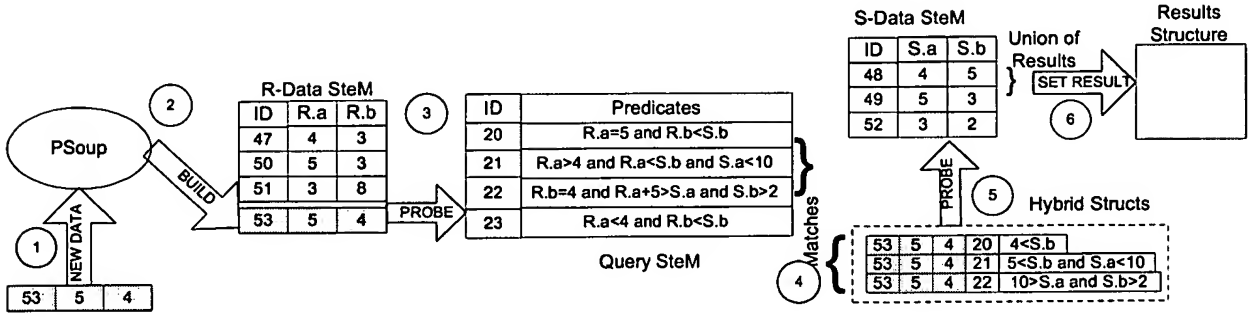


Figure 5: Join Processing: Entry of New R Tuple

This entry is then marked to reflect that this pair satisfied the specific queryID (Step 6).

Now consider the entry of a new R data tuple into the system (Figure 5). It is inserted into the R Data SteM, and first probes the Query SteM. The rest of the processing closely parallels the description for the entry of a new query above.

Observe that there is redundancy among the hybrid structs (the shaded parts of the structs in the figures). The new SQC tuple is repeated across all the hybrid structs in Figure 4 and similarly, the new data tuple is repeated across all the hybrid structs in Figure 5. This results in repeated computation in the probes of Step 5. This redundancy and techniques to remove it are described in detail in Section 5.4.

3.3 Query Invocation and Result Construction

In this section, we describe the Results Structure, and the processing performed by PSoup to return the query results when a previously specified query is invoked.

The Results Structure maintains information about which tuples in the Data SteM(s) satisfied which SQCs in the Query SteM. For each result tuple of each query, it stores the tupleIDs and physicalIDs of all the constituent base tuples of the result tuple. The Results Structure is updated continuously during the query-data join described in Sections 3.2.1 and 3.2.2. The results of a query can be accessed by its queryID. In addition, the results are ordered and indexed by tuple timestamp (physicalID), for efficient retrieval of results within a time-window.

Consider a user request for the current result of a previously specified query. Recall from Section 3.2 that the BEGIN-END clauses of query specifications are stored in the WindowsTable. The clause is now retrieved from the table, and the current values of the endpoints of the input window are determined. By virtue of the background symmetric join processing in PSoup, all the data in the system has already been joined with the SQC of the query specification, and the results of the query-data join are present in the Results Structure. The PSoup Engine can therefore directly access this structure and apply the current input window of the query over its contents to retrieve the tupleIDs of the base tuples that make up the current result tuples. The actual tuples themselves can then be retrieved from the Data SteMs using the tupleIDs and returned to the client.

For single-stream queries, the retrieval of the current window from the timestamp of the result tuples is straightforward. For Join queries, the process is more difficult because the results are composed of multiple base tuples, each with its own timestamp. We describe this in Section 4. Projections are performed just-in-time when the query is invoked, concurrent with result construction. Duplicate elimination, if required, is also done at this point.

4 Implementation

In Section 3, we stepped through the basic framework of our solution using simple examples. Here, we describe the implementation of PSoup within the Telegraph system. The principal components of our solution are the N-relation symmetric join operator and the Results Structure.

At the heart of the N-relation symmetric join is an operator that inserts new data/queries into the appropriate storage structures, and then uses them to probe all the other storage structures. The storage structures themselves provide insert and probe methods over data/queries. The Eddy and SteM mechanisms [AH00, Ram01] provide a framework for adaptive n-relation symmetric joins. They were, however, designed in a different context. Eddies were originally conceived as a tuple router between traditional join operators. SteMs were proposed as data structures that could be shared between the different join operations. In effect, SteMs eliminate the join modules themselves, leaving Eddy as the active agent for effecting the join. However, neither were SteMs designed to store queries, nor were Eddies designed to route them. In addition, the simultaneous evaluation of multiple standing queries, and the storage of the results requires the tracking of more state. The changes needed in Telegraph to support additional functionality in PSoup are described below.

4.1 Eddy

The Eddy performs its work by picking up the next data tuple to route from a queue called the *Tuple Pool*, and then sending it to one of many join operators according to its routing policy.

To allow the eddy to route SQCs and hybrid structs (in addition to data), all the entities are encoded as tuples. This is done by creating a "predicate attribute" to represent (possibly partially evaluated) queries, and having all tuples contain data and/or predicate attributes. In addition to the data and/or predicate attributes, each tuple also contains a "to-do" list (called the *Interest List*), that enumerates the SteMs that it remains to be routed through before the tuple can be considered completely processed. This list is the only interface between the tuple and the Eddy. The Eddy is thus oblivious to the underlying types of the tuples it routes. It picks the next destination of a tuple based only on the information in the tuple's Interest List.

There is, however, a subtle difference between the flavors of Eddy as described by Avnur and Hellerstein [AH00] and Madden et al. [MSHR02] and the PSoup Eddy. This leads to different semantics for the results output by the two systems for a given query.

We say that a query processor produces *Stream-Prefix Consistent* results if it atomically materializes the entire effects of processing an older tuple (data or query) in its output, before it materializes any of the effects of processing a newer tuple. At all times, the complete set of results materialized in the system are then identical to the results of completely executing some prefix of the query stream over some prefix of the data stream. This property serializes the effects of new tuples (query or data) in the order that they enter the system. Stream-Prefix Consistency is therefore the basis of our ability to support windowed queries over data streams.

The PSoup Eddy provides Stream Prefix Consistency by storing the new and temporary tuples separately in the *New Tuple Pool (NTP)* and the *Temporary Tuple Pool (TTP)* respectively. The PSoup eddy begins by picking a tuple from the NTP, and then processing all the temporary tuples in the TTP, before it picks another new tuple from the NTP. The use of a higher-priority tuple pool to store in-flight tuples serializes the effects of new tuples on the Results Structure in the order in which they enter the system, thus maintaining it in a *stream-prefix consistent state* at all times. The previous versions of Eddy cannot guarantee the Stream Prefix Consistency property. This is due to their use of a single Tuple Pool to store both new tuples and temporary (hybrid structs) tuples in-flight within a join query.

4.2 SteMs

SteMs are abstract data structures that provide insert and probe methods over their contents. PSoup implements the SteMs interface to store data and queries.[2] The performance of the SteMs would be highly inefficient if the data/queries were probed sequentially, and the boolean factors were tested individually in the manner described in Section 3. We therefore use indexes to speed up operations on data and queries.

4.2.1 Data SteM

Data SteMs are used to store and index the base data of a stream. There is one Data SteM for each stream that enters the system. Since PSoup supports range queries, we need a tree-based index for the data to provide efficient access to probing queries. There is one tree for every attribute of the stream. For our main memory based implementation, red-black trees were chosen because they are efficient and have low maintenance cost.

When a query probes the Data SteM, the different single-relation boolean factors of the query are used to probe the corresponding indexes, and the results of these probes are intersected to yield the final result. The technique used to intersect the individual probe results is similar to the one used in Query SteMs and is described in Section 4.2.2.

The Data SteM also maintains a hash-based index over tupleIDs for fast access during result construction.

4.2.2 Query SteM

Query SteMs are used to store and index queries. There is one Query SteM for the entire system, allowing sharing of work between queries that have different, but overlapping FROM clauses.

As with the data, it is desirable to index queries for quick (and shared) evaluation during probes. Numerous predicate indexes have been proposed in the literature [YG99, HCHK+99, SSH86, KKKK02]. We use an index similar to the one proposed in CACQ [MSHR02]: red-black trees are used to index the single-attribute single-relation boolean factors of a query. For every relation, there is one tree for boolean factors over each attribute that appears in an SQC. The trees are indexed by the constant c

[2] Since PSoup is currently implemented as a main-memory system, we restrict Data SteMs to only keep data within a certain maximum window specified as a system parameter. Supporting queries over data streams archived on disk is the subject of future work.

that appears in the expression ($R.a$ *RELOP* c). To support range predicates, the nodes of the red-black tree are enhanced as shown in Figure 6. Each node contains five arrays that store the queryIDs of the boolean factors that map to that node. There is one array for each relational operator ($<, <=, =, >=, >$).

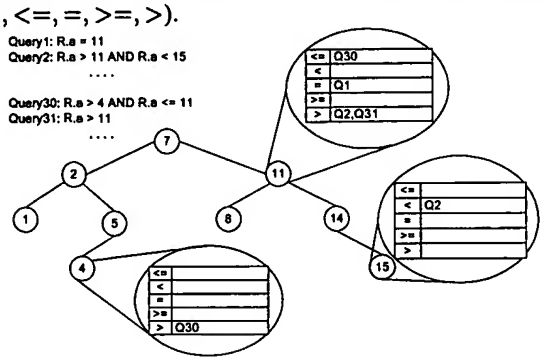


Figure 6: Predicate Index

To probe the query index using a data tuple r_i, an equality search is performed on the query index using the data value $r_i.a$ as the search key. The equality boolean factors that match the data are quickly identified by the node to which the search key maps. An index scan is then used to solve the inequality queries, if any.

The above expression for boolean factors only captures single-attribute boolean factors. Queries could also have multi-attribute selection or join boolean factors of the form ($R.a$ *RELOP* $[R.b|S.b][+/-c]$). Such boolean factors are not indexed, but are all stored in a single linked list called the predicateList.

Because a query can be split across the different predicate indexes and the predicateList, we need a technique for ANDing the results of the probes of these different structures. To do this, the Query SteM contains an array in which each cell corresponds to a query specification. At the beginning of a probe by a data tuple, the value of each cell is reset to the number of boolean factors in the corresponding query. Over the course of the various probes, every time the data tuple satisfies a boolean factor, the value of the corresponding cell in the array is decremented. A cell value of zero at the end of the probe indicates that the data tuple satisfied the query.

4.3 Results Structure

The last major component of our solution is the Results Structure, which is accessed when a user invokes a query to retrieve the current results for that query.

The Results Structure stores metadata that indicates which tuples satisfied which SQCs. Since the current main-memory implementation of PSoup only stores data within a certain maximum window, the results corresponding to expired data (and queries that have been removed from the system) are dropped. We use two different implementations of the Results Strucuture. One implementation (as described in Section 3) is a two-dimensional bitmap. There is a separate bitmap for each FROM-list that appears in any of the SQCs. The rows of this bitmap are ordered by the timestamp (physicalID) of the data. The columns are ordered by the ID of query. Indexes are provided over both the physicalID and the queryID.

The second implementation of the Results Structure associates with each query a linked list containing the data tuples that have satisfied it. The decision between the alternate structures can be made according to the tradeoff between the storage requirements of a (possibly) sparse bitmap, and a dense linked list.

As mentioned above, the results are sorted and indexed by tuple timestamp to speed up the application of the input window at query invocation. This is straightforward for single-table queries whose result tuples each have a single timestamp. The results of Join queries are, on the other hand, composed of multiple base tuples, each having its own timestamp. Only two of these timestamps, however, are significant: the earliest or the latest, since they serve to bound the age of the result tuples. The Results Structure associates only these two timestamps with each result tuple. The question arises as to which of the two timestamps (the earliest and the latest) should be used to sort and index the results. We expect queries typically to be landmark or sliding queries whose END clause (the later edge of the window) is defined as "NOW". All data tuples in the system have to be older than "NOW". As a result, the later edge of the window will not, in the common case, filter out any results. Therefore, the older timestamp is likely to be more significant for efficient result retrieval and is used to order the results.

We have now described how PSoup implements the duality of queries and data to apply new queries to old data, and new data to old queries. Now, we will describe its performance.

5 Performance

In this section, we investigate the performance of PSoup, focusing on the query invocation and data arrival rates supported by the system under different query workloads and input window sizes.

As mentioned earlier, PSoup is a part of the Telegraph project, and as such, it uses and extends the concept of Eddies and SteMs. However, because of the need to encode queries as tuples, and the difference in mechanisms for ANDing boolean factors in PSoup and CACQ, the tuple format in PSoup differs from both the formats used in the non-CQ version of Telegraph [HFCD+00], and CACQ [MSHR02]. Hence, we implemented new versions of both Eddy and SteMs. Like the rest of the Telegraph system, PSoup is implemented in Java.

In this section, we examine the performance of two different implementations of the system: *PSoup-partial (PSoup-P)* and *PSoup-complete (PSoup-C)*. PSoup-P is the implementation we have described in earlier sections: the results corresponding to the SQCs are maintained in the Results Structure, and the BEGIN-END clauses are applied to retrieve the current results on query invocation. PSoup-C on the other hand, continuously maintains the results corresponding to the current input window for each query

in linked lists. For comparison purposes, we also include measurements of a system (NoMat) that does not materialize results, but rather, executes each query from scratch when it is invoked. NoMat uses the same indices over the data and queries as the PSoup systems. When a query contains more than one boolean factor, we fix the order of probes over the data for NoMat such that the more selective boolean factors are applied first.

5.1 Storage Requirements

Before turning to the experiments, it is useful to examine the storage requirements of each system.

NOMAT: The storage cost is equal to the the space taken to store the base data streams within the maximum window over which queries are supported, plus the size of the structures used to store the queries themselves.

PSOUP-PARTIAL: In addition to costs incurred by NoMat, PSoup-P also pays the cost of the Results Structure, which uses either a bitarray or a linked-list to store the results, depending on whichever takes less storage. The cost of the first option depends on the number of standing queries stored in the system, and the maximum window over which queries can be asked. The cost of the latter approach depends on the result sizes (before the imposition of the time window). For the set of experiments described below, we chose the bitarray implementation for the PSoup-P Results Structure.

PSOUP-COMPLETE: Like PSoup-P, PSoup-C pays for the cost of storing the results in addition to the costs paid by NoMat systems. PSoup-C always stores the current results of standing queries at a given time. Under normal loads, we expect PSoup-C to have substantially higher storage requirements than PSoup-P which uses a dense bitarray.

5.2 Computational performance

The environment for which we have targeted PSoup is one in which new query specifications arrive much less frequently than the rate at which existing query specifications are invoked. We are therefore primarily concerned with the query invocation rate that can be supported in the system. We determine this rate by measuring the response time per query invocation for varying input window size and query complexity. We also wish to measure the maximum data arrival rate supported by the system. This maximum rate depends on the relative costs of the computation devoted to processing the entry of new data tuples, and the computation spent on maintaining the windows on the results that have been generated. A server is saturated by these two costs at the maximum data arrival rate that it can support.

There is an inherent tradeoff between result-invocation and data arrival rates. Lazy evaluation (as used in NoMat) suffers from poor response time while having no maintenance costs. Eager evaluation (as done in PSoup-C) offers excellent response time but has increased maintenance costs. PSoup-P eagerly evaluates the WHERE clause of its query specifications, but adopts a lazy approach with respect to the imposition of the time windows specified in the BEGIN-END clause. Its performance therefore lies between that of the other approaches.

5.2.1 Experimental setup

As mentioned in Section 5, we implemented PSoup in Java. In order to evaluate its performance we ran a number of experiments that varied the window sizes and the number and type of boolean factors (equality/inequality, single-relation, two-relation) of the queries, and measured the response time for query invocations under these different conditions. In addition to the response time for query invocations, we also looked at the maximum data arrival rate that can be supported by the system. We compared the maximum data arrival rates supported by two implementations of both PSoup-P and PSoup-C, one each with and without the use of predicate indexes. We also studied a scheme to remove a type of redundancy that arises in join processing (as was described in Section 3), and measured its performance under different workloads.

All the experiments were run on an unloaded server with two Intel PentiumIII, 666MHz, 256 KB on-chip cache processors. It had 768MB RAM. PSoup was run completely in main-memory, so we are not concerned with disk space. We use Sun's Java Hotspot(TM) Client VM, version "1.3.0", on Linux with a 2.2.16 kernel.

We used synthetically generated query and data streams to compare the three approaches under a range of application scenarios. The data values are uniformly distributed in the interval $[0, 255]$. In order to stress the system, we make all the tuples in the stream available instantaneously, i.e., there is no variable delay between consecutive tuples in the stream. Madden et al. [MSHR02] demonstrated the advantages of adaptive query processing gained by applying the Eddies framework to CQ processing. Those results also apply to this setting.

Parameters	Range of Values
Input Window Size (in #tuples)	2^7-2^{16}
#Query Specifications	2^7-2^{12}
#Boolean Factors	1-8

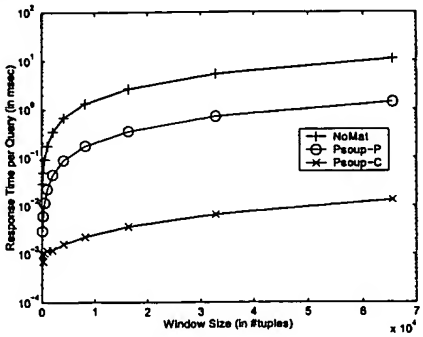
Table 1: Independent Parameters for Experiments

For single-relation boolean factors of the form (R.a RELOP c), the value of the constant c is chosen uniformly from among a multiple of 32 in the interval [0,255] with a probability of 0.2, and uniformly from the entire range $[0, 255]$ with probability 0.8. We used this multimodal distribution to approximate a query workload in which some items were more interesting than others. Join queries have exactly one multiple-relation boolean factor. This is done to isolate the effects of the join. The multiple-relation boolean factors are of the form (R.a RELOP S.b +/- c), where c has the same distribution as for Selection Queries.

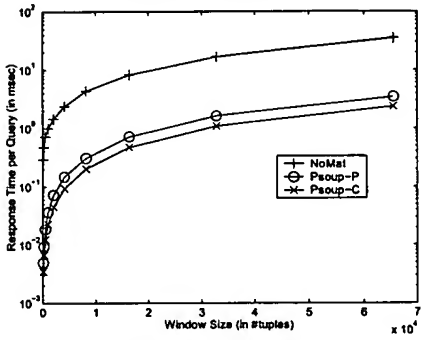
5.2.2 Response time vs window size

The first set of experiments we describe measures the time taken to respond to Select and Join query invocations with increasing input window sizes. Figure 7(a) shows the response time per query for selection queries with equality predicates, Figure 7(b) shows the same metric for selec-

tion queries with interval predicates. Interval predicates were formed by combining two single-relation inequality boolean factors over the same attribute (the size of the interval is uniformly distributed in the range $[0, 255]$). Note that the y-axes on both plots, use a logscale and the values on the x-axes have a multiplicative factor of 10,000. In both workloads, the queries have between one and four predicates.



(a) Equality Predicates



(b) Interval Predicates

Figure 7: Response-Time for Select Queries

The response times increase for all three systems with increasing window sizes. For NoMat, this is because of increased query execution time. For PSoup-P, this is caused by the increase in the length of the bitarray in the Results Structure. For PSoup-C, this is because of the increase in the cardinality of the results.

As expected, the response time for both workloads under NoMat is much worse than the other two. PSoup-P performs worse than PSoup-C by two orders of magnitude for equality queries because of the need to traverse an entire bitarray of the size of the maximum input window for each query, irrespective of the size of the result. For the same reason, the performance of PSoup-P does not change between equality and inequality queries, while the response time for both the NoMat and PSoup-C solutions are higher for inequality queries than equality queries - the former because of greater data index traversal, the latter because of larger result sets. The performance of PSoup-P and PSoup-C is comparable for inequality queries.

Figure 8 shows the response time for two-table inequality Join queries with varying input window size. In this case, the y-axis uses a linear scale. The x-axis shows the window size for each table of the join. The result size aggregated over all queries is proportional to the square of the window size. The range of the window size is therefore much smaller than for Selection queries. The response time for NoMat is about two orders of magnitude worse than that of the PSoup systems. PSoup-P is less than an order of magnitude worse than PSoup-C. For example, at a window size of 576 , the response time for PSoup-P is 29.98 msec, while for PSoup-C it is 8.54 msec.

We conclude from this experiment that as expected, systems that do not materialize the results of the queries do not not scale with increasing input window size.

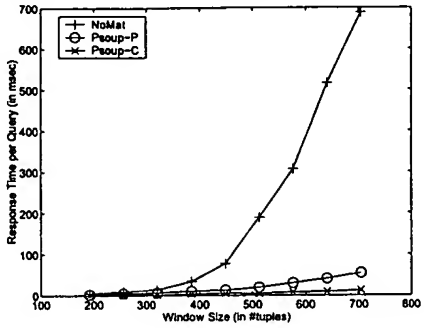


Figure 8: Response-Time for Join Queries

5.2.3 Response time vs #Interval predicates

The next experiment we describe measures the response time for inequality Selection queries as we vary the number of conjoined interval predicates in the query. The queries contain one interval predicate over each attribute that appeared in the SQC. The input window size was fixed at 2^{15} (the second largest window size shown in Figure 7 for Selection queries) for all the queries. The results are shown in Table 2.

As expected, both of the PSoup solutions again outperform NoMat by between one to two orders of magnitude (according to the number of interval predicates in the query). An interesting point to note is that while the response time for NoMat increases with the number of conjoined interval predicates due to the greater amount of computation required, the response times for PSoup-P and PSoup-C *decrease* significantly. The behavior of NoMat is explained by the increasing complexity of the queries that have to be executed upon invocation by the user. The relative performance of the PSoup implementations is explained by the cost of result construction in the two systems. Whereas PSoup-C constructs its results by dereferencing pointers to the data tuples stored in its linked list and then copying the tuples, PSoup-P has to pay the extra cost first retrieving the references to the data tuples using the Data SteM's index over physicalIDs. The fact that both of their response times reduce with increasing number of ANDed interval predicates is because of the higher selectivity of the resulting queries and the correspondingly smaller result sizes.

Another interesting point is the switch in the relative performance of PSoup-P and PSoup-C as we go from one

#Interval Predicates	Response Time (in msec)		
	NoMat	PSoup-P	PSoup-C
1	0.3940	0.0465	0.0565
2	0.4905	0.0240	0.0210
4	0.8255	0.0130	0.0035

Table 2: Response Time: Selection w/ Interval Predicates

to two interval predicates. This is explained as follows. For queries with one interval predicate, the selectivity of the query is poor so that the relative inefficiency of linked-list traversal in PSoup-C compared to bitarray traversal in PSoup-P outweighs the fact that fewer elements have to be traversed. With increasing number of interval predicates however, the selectivity increases and the difference in the average size of the result sets and the input window (2^{15}) becomes pronounced enough to dominate the relative costs.

In conclusion, this experiment shows that NoMat does not scale with increasing query complexity. Both PSoup implementations have comparable performance for fewer boolean factors, but PSoup-C's performance improves dramatically due to the reduction in result sizes for more selective queries.

5.2.4 Data arrival rate vs #SQCs

We now turn our attention to the maximum data arrival rates supported by PSoup with varying number of inequality Selection query specifications in system. We do not consider NoMat for this experiment. We consider two possible implementations of both PSoup-P and Psoup-C: one each with and without predicate indexes (referred to as Shrd and Unshrd respectively). The comparison of PSoup-P and PSoup-C highlights the effect of lazy vs. active maintenance of results on the data arrival rates. The difference in the performance of versions of PSoup using predicate indexes with those that do not highlights the savings in computation achieved through the use of predicate indexes.

A fully loaded server either keeps the query results current, or accepts new data. The relative costs of the two activities therefore help us determine the maximum data rate that can be supported by the system for a given number of stored query specifications. The window size for all the query specifications in this experiment is fixed at 1000 tuples.

The results for this experiment are shown in Figure 9. The y-axis uses a logscale. The PSoup-P_Shrd solution performs the best, and beats the PSoup-P_Unshrd system by an order of magnitude and the two PSoup-C based solutions by two orders of magnitude. It is interesting to note that the cost of maintaining the results dominates the cost of incremental computation upon entry of new data to the extent that it almost does not matter whether or not we share the computation though indexing queries for the PSoup-C implementations. This indicates that if we wish to support high data arrival rates, PSoup-P_Shrd is the implementation of choice. An interesting result in this experiment is that the speedup achieved by PSoup-P_Shrd over PSoup-P_Unshrd through the use of query indexes increases with increasing number of query specifications. This happens because the boolean factors of new query specifications increasingly fall into the old nodes in the predicate index, thus keeping the computation amount roughly the same.

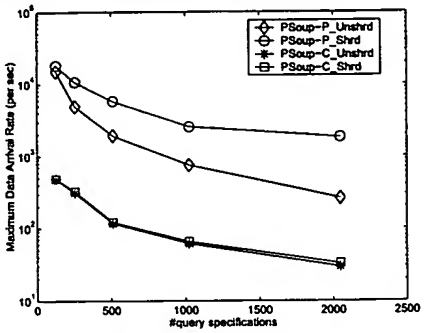


Figure 9: Data Arrival Rate for Selection Queries

This experiment confirms our expectation that the decision not to index queries, and to maintain query results up to date, can both adversely affect the data arrival rate that can be supported in the system.

5.3 Summary of Results

The first two experiments demonstrate that materializing the results of queries allows the support of higher query invocation rates. The third experiment shows us that indexing queries and lazily applying the windows improves the maximum data throughput supported by the system. The choice between the PSoup-P and PSoup-C implementations thus depends on the amount of memory we have in the system (PSoup-C requires more), and whether we wish to optimize for query invocation rate (PSoup-C) or data arrival rate (PSoup-P).

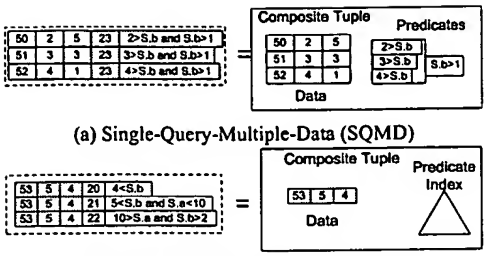
5.4 Removing redundancy in join processing

As mentioned in Section 3, the join processing discussed so far can perform redundant work. In this section, we will describe the redundancy, and show how we overcome it.

5.4.1 Entry of a query specification or new data

Recall from Section 3, the production of hybrid structs in the processing of new query specifications. The relevant part of Figure 4, which detailed the processing of a new Join query (R.a<5 and R.a>S.b and S.b>1) over streams R and S, is reproduced in Figure 10(a) for convenience. The hybrid structs that are produced after the query specification probes the R-Data SteM share the same *S-only* component (S.b > 1) of the original query. This boolean factor repeatedly probes the S-Data SteM (once for each hybrid struct). We can eliminate this redundancy by combining all the hybrid tuples produced by the probe of the RS query into the R-Data SteM into a single *"single query-multiple data"* composite tuple (Figure 10(a)). The shared *S-only* component can now be applied exactly once. More interestingly, we can use a sort-merge join based approach to join the set of predicates with the set of tuples in the S-Data SteM.

A similar situation arises when data is added to the system. The hybrid structs produced during the processing of



(a) Single-Query-Multiple-Data (SQMD)

(b) Single-Data-Multiple-Query (SDMQ)

Figure 10: Join Redundancy - Composite Tuples

the new data share many boolean factors. The relevant part of Figure 5 is reproduced in Figure 10(b). The identical boolean factors are executed repeatedly over the same data set in the S-Data SteM. The *"single data-multiple query"* composite tuple (Figure 10(b)) can be used in conjunction with the sort-merge join based approach to apply the composite tuple to the Data SteM.

5.4.2 Composite tuples in joins

This experiment compares the costs of incremental computation on arrival of a new Join query specification over streams R and S, with and without the use of composite tuples. The execution path for the new query specification is the same as was shown in Figure 4.

The Join queries are of the form: (R.a RELOP1 c1) AND (R.a RELOP2 S.b) AND (S.b RELOP3 c2). To isolate the effect of the composite tuple from the other steps involved in join processing, we only measure the cost of Step 5 of the join processing shown in Figure 4. After executing Steps 1 through 3 of Figure 4, the query predicates are of the form (R.a_value RELOP2 S.b) AND (S.b RELOP3 c2). The latter boolean factor is shared across all hybrid structs. We now compare the cost of probing the S-Data SteM with the composite tuples against the cost of probing it with the individual hybrid tuples.

By varying RELOP2 and RELOP3, we create three different workloads. In the first, we set RELOP3 to be 'equals' (Eq), and RELOP2 to be one of the inequalities (Ineq). In the second workload, we reverse this. In the final workload, we set both to be inequalities.

The results are shown in Figure 11. The legend in the plot reflects the choices for RELOP2/RELOP3 (Equality or Inequality), and whether composite tuples were used (Composite) or not (Separate). Note that the y-axis uses a logscale. For the Eq/Ineq workload, the shared boolean factor is an equality and is therefore highly selective (because of the uniform distribution of the data). Hence, in both approaches, it is applied first, before any other boolean factors are used to probe the data. The solution using the composite tuple probes the data with this factor exactly once, effectively halving the total number of boolean factors that eventually probe the data. It is therefore approximately twice as efficient as the other approach. For the Ineq/Eq workload, it does not help much to apply the shared inequality factor first. However, the composite tuple based approach using a Sort-Merge join of the boolean factors and data still outperforms the other approach using Nested Loops, because most of the boolean factors are equality factors, and Sort-Merge is a more efficient algorithm for equijoins than nested loops. In the Ineq/Ineq workload however, both the shared and the individual boolean factors are inequalities. Sort-Merge is not a good algorithm for inequality joins, therefore the Nested Loops index join solution is preferred.

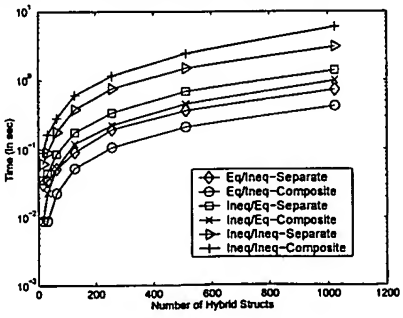


Figure 11: Probe times w/ and w/o composite tuples

6 A Note on Aggregation Queries

To this point, we have only discussed SPJ queries, but PSoup also supports aggregates such as count, sum, average, min and max.

Ideally, we would like to share the data structures used in computing aggregates across repeated invocations of all SELECT-PROJECT-JOIN-AGGREGATE queries over streams, just as we do in the case of SELECT-PROJECT-JOIN queries. However, it is only possible to share these data structures across queries that have the same SELECT-PROJECT-JOIN clause.

We demonstrate the above claim using example queries that compute the MAX of the results of a SELECT-FROM-WHERE query. First, we explain the basic approach to computing the MAX over different windows using the same data structure. Consider Figure 12, which shows a ranked n-ary tree over all the data in a SteM. The leaves of the tree

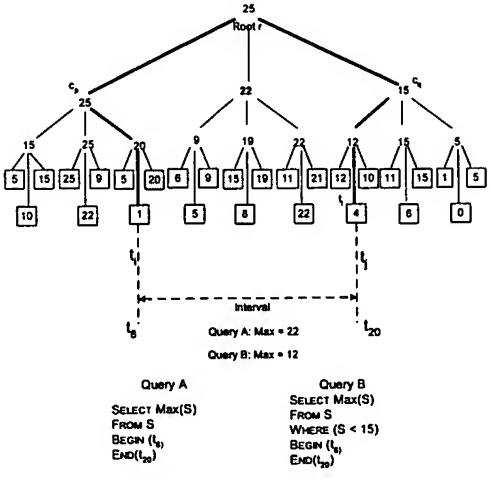


Figure 12: Ranked Tree for Max

are ordered by time of insertion into the SteM (i.e., insertions always occur at the rightmost node of the tree). Each

node is annotated with the MAX of all the elements under the subtree rooted at that node.

Now, let us invoke query A on the system when the current window is $[t_i, t_j]$. The first common ancestor of the end points of the window is the root r. Let c_p and c_q be the children of r that need to be followed to reach t_i and t_j. Let the rightmost leaf under the subtree rooted at c_p be t_k and the leftmost leaf under the subtree rooted at c_q be t_l. $Max[r, t_i, t_j] = Max(Max[c_p, t_i, t_k]$, *annotations of all children of* r *between* c_p *and* c_q, $Max[c_q, t_l, t_j])$.

This is a recursive expression and can be computed in $O(\log n)$ time by following the nodes of the tree down to t_i and t_j. In the figure, the thick edges show the paths traversed in this recursion in the specific case where $[t_i, t_j] \equiv [t_8, t_{20}]$; the maximum is 22.

Now consider Query B. It has a different SELECT-FROM-WHERE clause from Query A, and the values 20, 15, 19, 22 and 21 are *not* to be considered computing Query B. This tree can therefore not be used directly to compute Query B. The problem is that the leaves in the tree match the results of the SELECT-FROM-WHERE clause of Query A but not Query B. Therefore, we must maintain a separate structure for each in the Query SteM. Sharing occurs only between different invocations of the same query.

7 Conclusion

In conclusion, we have described the design and implementation of a novel query engine that treats data and query streams analogously and performs multiquery evaluation by joining them. This allows PSoup to support queries that require access to both data that arrived prior to the query specification, and also data that appears after. PSoup also separates the computation of the results from their delivery by materializing the results: this allows PSoup to support disconnected operation. These two features enable data recharging and monitoring applications that intermittently connect to a server to retrieve the results of a query. We also describe techniques for sharing both computation and storage across different queries.

In terms of future work, there is much to be done. Psoup is currently implemented as a main memory system. We would like to be able to archive data streams to disk and support queries over them. Disk based stores raise the possibility of swapping not only data, but also queries between disk and main memory. Swapping queries out of main memory would effectively deschedule them, and can be used as a scheduling mechanism if some queries are invoked much more frequently than others. In this paper, we have only briefly touched upon the relation of registered queries in PSoup to materialized views. We intend to further explore the space of materialized views over infinite streams, especially under resource constraints. The current implementation of PSoup allows the client only to retrieve answers corresponding to the current window. We intend to relax this restriction, and allow clients to treat PSoup more generally as a query browser for temporal data.

References

[AF00] ALTINEL, M., AND FRANKLIN, M. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB* (2000), pp. 53-64.

[AFZ01] AKSOY, D., FRANKLIN, M., AND ZDONIK, S. Data Staging for On-Demand Broadcast. In *VLDB* (2001), pp. 571-580.

[AH00] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously Adaptive Query Processing. In *SIGMOD* (2000), pp. 261-272.

[BGS01] BONNET, P., GEHRKE, J., AND SESHADRI, P. Towards Sensor Database Systems. In *ICMDM* (2001), pp. 3-14.

[BS00] BONNET, P., AND SESHADRI, P. Device Database Systems. In *ICDE* (2000), pp. 194.

[BW01] BABU, S., AND WIDOM, J. Continuous Queries over Data Streams. *SIGMOD Record* (September 2001), 109-120.

[CCCC+02] CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Monitoring Streams - A New Class of Data Management Applications. In *VLDB* (2002).

[CDTW00] CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD* (2000), pp. 379-390.

[CFZ01] CHERNIACK M., FRANKLIN, M., AND ZDONIK, S. Expressing User Profiles for Data Recharging. In *IEEE Personal Communications* (August, 2001), pp. 6-13.

[DGIM02] DATAR, M., GIONIS, A., INDYK, P. AND MOTWANI, R. Maintaining Stream Statistics over Sliding Windows. In *ACM-SIAM SODA* (2002).

[F82] FORGY, C. Rete: A Fast Algorithm For the Many Patterns/Many Objects Match Problem. In *Artifical Intelligence* (1982), 19(1):pp. 17-37.

[FJLP+01] FABRET, F., JACOBSEN, H., LLIBRAT, F., PEREIRA, J., ROSS, K., AND SHASHA D. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD* (2001), pp 115-126.

[FGCB+97] FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. Cluster-Based Scalable Network Services. In *SOSP* (1997), pp. 78-91.

[GKS01] GEHRKE, J., KORN, F., AND SRIVASTAVA, D. On Computing Correlated Aggregates over Continual Data Streams. In *SIGMOD* (2001), pp. 13-24.

[HBC97] HANSON, E., BODAGALA, S., AND CHADAGA, U. Optimized Trigger Condition Testing in Ariel Using Gator Networks. University of Florida CISE Department Tech. Report TR97-021, Nov. 1997.

[HCHK+99] HANSON, E., CARNES, C., HUANG, L., KONYALA, M., NORONHA, L., PARTHASARATHY, S., PARK, J., AND VERNON, A. Scalable Trigger Processing. In *ICDE* (1999), pp. 266-275.

[HFCD+00] HELLERSTEIN, J., FRANKLIN, M., CHANDRASEKARAN, S., DESHPANDE, A., HILDRUM, K., MADDEN, S., RAMAN, V., AND SHAH, M. Adaptive Query Processing: Technology in Evolution. In *IEEE Data Engg. Bulletin*. March 2000. pp. 7-18.

[JMS95] JAGADISH, H., MUMICK, I., AND SILBERSCHATZ, A. View Maintenance Issues for the Chronicle Data Model. In *PODS* (1995), pp. 113-124.

[KKKK02] KEIDL, M., KREUTZ, A., KEMPER, A., AND KOSSMANN, D. A Publish & Subscribe Architecture for Distributed Metadata Management. In *ICDE* (2002), pp. 309-320.

[LSM99] LEE, W., STOLFO, S., AND MOK, K. Mining in a Data-flow Environment: Experience in Network Intrusion detection. In *SIGKDD* (1999), pp. 114-124.

[M87] MIRANKER, D. TREAT: A Better Match Algorithm for AI Production System Matching. In *AAII* (1987), pp. 42-47.

[MF02] MADDEN, S., AND FRANKLIN, M. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *ICDE* (2002)

[MSHR02] MADDEN, S., SHAH, M., HELLERSTEIN, J., AND RAMAN, V. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD* (2002).

[OQ97] O'NEIL, P., AND QUASS, D. Improved Query Performance with Variant Indexes. In *SIGMOD* (1997), pp. 38-49.

[Ram01] RAMAN, V. *Interactive Query Processing*. PhD thesis, UC Berkeley, 2001.

[SG97] SHIVAKUMAR, N., AND GARCIA-MOLINA, H. Wave-Indices: Indexing Evolving Databases. In *SIGMOD* (1997), pp. 381-392.

[SH98] SULLIVAN, M., AND HEYBEY, A. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX* (1998).

[SLR94] SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. Sequence Query Processing. In *SIGMOD* (1994), pp. 430-441.

[SSH86] STONEBRAKER M., SELLIS T. K., AND HANSON E. N. An Analysis of Rule Indexing Implementations in Data Base Systems In *Expert Database Conf.* (1986) pp. 465-476

[SWCD97] SISTLA, A., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. Modeling and Querying Moving Objects. In *ICDE* (1997), pp. 422-432.

[SZZA01] SADRI, R., ZANIOLO, C., ZARKESH, A., AND ADIBI, J. Optimization of Sequence Queries in Database Systems. In *PODS* (2001), pp. 71-81.

[TGNO92] TERRY, D., GOLDBERG, D., NICHOLS, D,, AND OKI, B. Continuous Queries over Append-Only Databases. In *SIGMOD* (1992), pp. 321-330.

[UFA98] URHAN, T., FRANKLIN, M., AND AMSALEG, L. Cost Based Query Scrambling for Initial Delays. In *SIGMOD* (1998), pp. 130-141.

[UF00] URHAN, T., AND FRANKLIN, M. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin* (2000) 23(2):pp. 27-33.

[WA91] WILSCHUT A., AND APERS, P. Dataflow Query Execution in a Parallel Main-memory Environment. In *PDIS* (1991), pp. 68-77.

[YG99] YAN T. W., AND GARCIA-MOLINA, H. The SIFT Information Dissemination System. In *TODS* (1999), pp. 529-565.

Monitoring Streams – A New Class of Data Management Applications

Don Carney
Brown University
dpc@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Christian Convey
Brown University
cjc@cs.brown.edu

Sangdon Lee
Brown University
sdlee@cs.brown.edu

Greg Seidman
Brown University
gss@cs.brown.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Abstract

This paper introduces monitoring applications, which we will show differ substantially from conventional business data processing. The fact that a software system must process and react to continual inputs from many sources (e.g., sensors) rather than from human operators requires one to rethink the fundamental architecture of a DBMS for this application area. In this paper, we present Aurora, a new DBMS that is currently under construction at Brandeis University, Brown University, and M.I.T. We describe the basic system architecture, a stream-oriented set of operators, optimization tactics, and support for real-time operation.

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions on this repository. We call this a *Human-Active, DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrives asynchronously and answers must be computed with incomplete information. Lastly, DBMSs assume that applications require no real-time services.

There is a substantial class of applications where all five assumptions are problematic. Monitoring applications are applications that monitor continuous streams of data. This class of applications includes military applications that monitor readings from sensors worn by soldiers (e.g., blood pressure, heart rate, position), financial analysis applications that monitor streams of stock data reported from various stock exchanges, and tracking applications that monitor the locations of large numbers of objects for which they are responsible (e.g., audio-visual departments that must monitor the location of borrowed equipment). Because of the high volume of monitored data and the query requirements for these applications, monitoring applications would benefit from DBMS support. Existing DBMS systems, however, are ill suited for such applications since they target business applications.

First, monitoring applications get their data from external sources (e.g., sensors) rather than from humans issuing transactions. The role of the DBMS in this context is to alert humans when abnormal activity is detected. This is a *DBMS-Active, Human-Passive* (*DAHP*) model.

Second, monitoring applications require data management that extends over some history of values reported in a stream, and not just over the most recently reported values. Consider a monitoring application that tracks the location of items of interest, such as overhead transparency projectors and laptop computers, using electronic property stickers attached to the objects. Ceiling-mounted sensors inside a building and the GPS system in the open air generate large volumes of location data. If a reserved overhead projector is not in its proper location, then one might want to know the geographic position of the missing projector. In this case, the last value of the monitored object is required. However, an administrator might also want to know the duty cycle of the projector, thereby requiring access to the entire historical time series.

Third, most monitoring applications are trigger-oriented. If one is monitoring a chemical plant, then one wants to alert an operator if a sensor value gets too high or if another sensor value has recorded a value out of range more than twice in the last 24 hours. Every application could potentially monitor multiple streams of data, requesting alerts if complicated conditions are met. Thus, the scale of

† This work was supported by the National Science Foundation under NSF Grant number IIS00-86057 and a gift from Sun Microsystems.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

trigger processing required in this environment far exceeds that found in traditional DBMS applications.

Fourth, stream data is often lost, stale, or intentionally omitted for processing reasons. An object being monitored may move out of range of a sensor system, thereby resulting in lost data. The most recent report on the location of the object becomes more and more inaccurate over time. Moreover, in managing data streams with high input rates, it might be necessary to shed load by dropping less important input data. All of this, by necessity, leads to approximate answers.

Lastly, many monitoring applications have real-time requirements. Applications that monitor mobile sensors (e.g., military applications monitoring soldier locations) often have a low tolerance for stale data, making these applications effectively real time. The added stress on a DBMS that must serve real-time applications makes it imperative that the DBMS employ intelligent resource management (e.g., scheduling) and graceful degradation strategies (e.g., load shedding) during periods of high load. We expect that applications will supply Quality of Service (QoS) specifications that will be used by the running system to make these dynamic resource allocation decisions.

Monitoring applications are very difficult to implement in traditional DBMSs. First, the basic computation model is wrong: DBMSs have a HADP model while monitoring applications often require a DAHP model. In addition, to store time-series information one has only two choices. First, he can encode the time series as current data in normal tables. In this case, assembling the historical time series is very expensive because the required data is spread over many tuples, thereby dramatically slowing performance. Alternately, he can encode time series information in binary large objects to achieve physical locality, at the expense of making queries to individual values in the time series very difficult. One system that tries to do something more intelligent with time series data is the Informix Universal Server, which implemented a time-series data type and associated methods that speed retrieval of values in a time series [2]; however, this system does not address the concerns raised above.

If a monitoring application had a very large number of triggers or alerters, then current DBMSs would fail because they do not scale past a few triggers per table. The only alternative is to encode triggers in some middleware application. Using this implementation, the system cannot reason about the triggers (e.g., optimization), because they are outside the DBMS. Moreover, performance is typically poor because middleware must poll for data values that triggers and alerters depend on.

Lastly, no DBMS that we are aware of has built-in facilities for approximate query answering. The same comment applies to real-time capabilities. Again, the user must build custom code into his application.

For these reasons, monitoring applications are difficult to implement using traditional DBMS technology. To do better, all the basic mechanisms in current DBMSs must be rethought. In this paper, we describe a prototype system, *Aurora*, which is designed to better support monitoring applications. We use Aurora to illustrate design issues that would arise in any system of this kind.

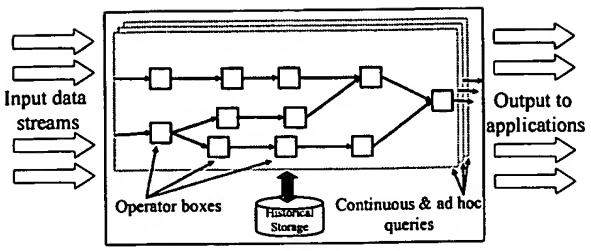


Figure 1: Aurora system model

Monitoring applications are applications for which streams of information, triggers, imprecise data, and real-time requirements are prevalent. We expect that there will be a large class of such applications. For example, we expect the class of monitoring applications for physical facilities (e.g., monitoring unusual events at nuclear power plants) to grow in response to growing needs for security. In addition, as GPS-style devices are attached to a broader and broader class of objects, monitoring applications will expand in scope. Currently such monitoring is expensive and is restricted to costly items like automobiles (e.g., Lojack technology). In the future, it will be available for most objects whose position is of interest.

In Section 2, we begin by describing the basic Aurora architecture and fundamental building blocks. In Section 3, we show why traditional query optimization fails in our environment, and present our alternate strategies for optimizing Aurora applications. Section 4 describes the run-time architecture and behavior of Aurora, concentrating on storage organization, scheduling, introspection, and load shedding. In Section 5, we discuss the myriad of related work that has preceded our effort. We describe the status of our prototype implementation in Section 6, and conclude in Section 7.

2 Aurora System Model

Aurora data is assumed to come from a variety of data sources such as computer programs that generate values at regular or irregular intervals or hardware *sensors*. We will use the term *data source* for either case. In addition, a data *stream* is the term we will use for the collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and Aurora timestamps every incoming tuple to monitor the quality of service being provided.

The basic job of Aurora is to process incoming streams in the way defined by an *application administrator*. Aurora is fundamentally a data-flow system and uses the popular *boxes and arrows* paradigm found in most process flow and workflow systems. Hence, tuples flow through a loop-free, directed graph of processing operations (i.e., *boxes*). Ultimately, output streams are presented to *applications*, which must be programmed to deal with the asynchronous

tuples in an output stream. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries. Figure 1 illustrates the high-level system model.

2.1 Operators

Aurora contains built-in support for eight primitive operations for expressing its stream processing requirements. Included among these are *windowed* operators that operate on sets of consecutive tuples from a stream ("windows") at a time. Every windowed operator applies an input (user-defined) function to a window and then advances the window to capture a new set of tuples before repeating the processing cycle. *Slide* advances a window by "sliding" it downstream by some number of tuples. This operator could be used to perform rolling computations, as in a query that continuously determines the average value of IBM stock over the most recent three hours. *Tumble* resembles Slide except that consecutive windows have no tuples in common. Rather, Tumble effectively partitions a stream into disjoint windows. This is useful, for example, when calculating daily stock indexes, where every stock quote is used in exactly one index calculation. *Latch* resembles Tumble but can maintain internal state between window calculations. This is useful for "infinite window" calculations, such as one that maintains the maximum or average value of every stock, maintained over its lifetime. Finally, *Resample* produces a partially synthetic stream by interpolating tuples between actual tuples of an input stream.

Aside from Aurora's windowed operations are operators that act on a single tuple at a time. The *Filter* operator screens tuples in a stream for those that satisfy some input predicate. A special case of Filter is *Drop*, which drops random tuples at some rate specified as an operator input. *Map* applies an input function to every tuple in a stream. *GroupBy* partitions tuples across multiple streams into new streams whose tuples contain the same values over some input set of attributes. Finally, *Join* pairs tuples from input streams whose "distance" (e.g., difference in timestamps) falls within some given upper bound. For example, this distance might be set to 30 minutes if one wanted to pair stocks whose prices coincide within a half-hour of each other.

Other desirable idioms for stream processing can be expressed as compositions of Aurora's built-in primitives. For example, while Aurora has no built-in "CASE statement" operator, one can be simulated by first applying a Map operator to a stream (that assigns a value to a new attribute that is dependent on which case predicate is satisfied) and then using GroupBy to partition tuples according to values assigned to this attribute. Additionally, there is no explicit Split box; instead a query can connect the output of one box to the inputs of several others.

A full treatment of these operators is beyond the scope of this paper.

2.2 Query Model

Aurora supports continual queries (real-time processing), views, and ad-hoc queries all using substantially the same

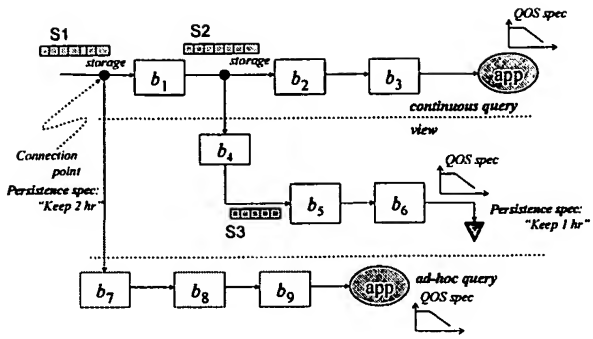


Figure 2: Aurora query model

mechanisms. All three modes of operation use the same conceptual building blocks. Each mode processes flows based on *QoS specifications*—each output in Aurora is associated with two-dimensional QoS graphs that specify the utility of the output in terms of several performance and quality related attributes (see Section 4.1). The diagram in Figure 2 illustrates the processing modes supported by Aurora.

The topmost path represents a *continuous query*. In isolation, data elements flow into boxes, are processed, and flow further downstream. In this scenario, there is no need to store any data elements once they are processed. Once an input has worked its way through all reachable paths, that data item is drained from the network. The QoS specification at the end of the path controls how resources are allocated to the processing elements along the path. One can also view an Aurora network (along with some of its applications) as a large collection of triggers. Each path from a sensor input to an output can be viewed as computing the *condition* part of a complex trigger. An output tuple is delivered to an application, which can take the appropriate action.

The dark circles on the input arcs to boxes b_1 and b_2 represent *connection points*. A connection point is an arc that will support dynamic modification to the network. New boxes can be added to or deleted from a connection point. When a new application connects to the network, it will often require access to the recent past. As such, a connection point has the potential for persistent storage (see Section 4.2). Persistent storage retains data items beyond their processing by a particular box. In other words, as items flow past a connection point, they are cached in a persistent store for some period of time. They are not drained from the network by applications. Instead, a persistence specification indicates exactly how long the items are kept. In the figure, the left-most connection point is specified to be available for *two hours*. This indicates that the beginning of time for newly connected applications will be two hours in the past.

The middle path in Figure 2 represents a *view*. In this case, a path is defined with no connected application. It is allowed to have a QoS specification as an indication of the importance of the view. Applications can connect to the end of this path whenever there is a need. Before this

happens, the system can propagate some, all, or none of the values stored at the connection point in order to reduce latency for applications that connect later. Moreover, it can store these partial results at any point along a view path. This is analogous to a materialized or partially materialized view. View materialization is under the control of the scheduler.

The bottom path represents an *ad-hoc query*. An ad-hoc query can be attached to a connection point at any time. The semantics of an ad-hoc query is that the system will process data items and deliver answers from the earliest time T (persistence specification) stored in the connection point until the query branch is explicitly disconnected. Thus, the semantics for an Aurora ad-hoc query is the same as a continuous query that starts executing at $t_{now} - T$ and continues until explicit termination.

2.3 Graphical User Interface

The Aurora user interface cannot be covered in detail because of space limitations. Here, we mention only a few salient features. To facilitate designing large networks, Aurora will support a hierarchical collection of groups of boxes. A designer can begin near the top of the hierarchy where only a few *superboxes* are visible on the screen. A *zoom* capability is provided to allow him to move into specific portions of the network, by replacing a group with its constituent boxes and groups. In this way, a browsing capability is provided for the Aurora diagram.

Boxes and groups have a tag, an argument list, a description of the functionality and ultimately a manual page. Users can *teleport* to specific places in an Aurora network by querying these attributes. Additionally, a user can place *bookmarks* in a network to allow him to return to places of interest.

These capabilities give an Aurora user a mechanism to query the Aurora diagram. The user interface also allows monitors for arcs in the network to facilitate debugging, as well as facilities for "single stepping" through a sequence of Aurora boxes. We plan a graphical performance monitor, as well as more sophisticated query capabilities.

3 Aurora Optimization

In traditional relational query optimization, one of the primary objectives is to minimize the number of iterations over large data sets. Stream-oriented operators that constitute the Aurora network, on the other hand, are designed to operate in a data flow mode in which data elements are processed as they appear on the input. Although the amount of computation required by an operator to process a new element is usually quite small, we expect to have a large number of boxes. Furthermore, high data rates add another dimension to the problem. Lastly, we expect many changes to be made to an Aurora network over time, and it seems unreasonable to take the network off line to perform a compile time optimization. We now present our strategies to optimize an Aurora network.

3.1 Dynamic Continuous Query Optimization

We begin execution of an unoptimized Aurora network; i.e., the one that the user constructed. During execution, we gather run time statistics, such as the average cost of box execution and box selectivity. Our goal is to perform run-time optimization of a network, without having to quiesce it. Hence, combining all the boxes into a massive query and then applying conventional query optimization is not a workable approach. Besides being NP-complete [23], it would require quiescing the whole network. Instead, the optimizer will select a portion of the network for optimization. Then, it will find all connection points that surround the subnetwork to be optimized. It will hold all input messages at upstream connection points and drain the subnetwork of messages through all downstream connection points. The optimizer will then apply the following local tactics to the identified subnetwork.

- *Inserting Projections.* It is unlikely that the application administrator will have inserted map operators to project out all unneeded attributes. Examination of an Aurora network allows us to insert or move such map operations to the earliest possible points in the network, thereby shrinking the size of the tuples that must be subsequently processed. Note that this kind of optimization requires that the system be provided with operator signatures that describe the attributes that are used and produced by the operators.
- *Combining Boxes.* As a next step, Aurora diagrams will be processed to combine boxes where possible. A pair-wise examination of the operators suggests that, in general, map and filter can be combined with almost all of the operators whereas windowed or binary operators cannot.

It is desirable to combine two boxes into a single box when this leads to some cost reduction. As an example, a map operator that only projects out attributes can be combined easily with any adjacent operator, thereby saving the box execution overhead for a very cheap operator. In addition, two filtering operations can be combined into a single, more complex filter that can be more efficiently executed than the two boxes it replaces. Not only is the overhead of a second box activation avoided, but also standard relational optimization on one-table predicates can be applied in the larger box. In general, combining boxes at least saves the box execution overhead and reduces the total number of boxes, leading to a simpler diagram.

- *Reordering Boxes.* Reordering the operations in a conventional relational DBMS to an equivalent but more efficient form is a common technique in query optimization. For example, filter operations can sometimes be pushed down the query tree through joins. In Aurora, we can apply the same technique when two operations commute.

To decide when to interchange two commutative operators, we make use of the following performance model. Each Aurora box, b, has a *cost*, $c(b)$, defined as the expected execution time for b to process one input tuple. Additionally, each box has a *selectivity*, $s(b)$, which is the

expected number of output tuples per input tuple. Consider two boxes, b_i and b_j, with b_j following b_i. In this case, for each input tuple for b_i, we can compute the amount of processing as $c(b_i) + c(b_j) \times s(b_i)$. Reversing the operators gives a like calculation. Hence, we can compute the condition used to decide whether the boxes should be switched as:

$$1 - s(b_j)/c(b_j) > 1 - s(b_i)/c(b_i)$$

It is straightforward to generalize the above calculation to deal with cases that involve fan-in or fan-out situations. Moreover, it is easy to see that we can obtain an optimal ordering by sorting all the boxes according to their corresponding ratios in decreasing order. We use this result in a heuristic algorithm that iteratively reorders boxes (to the extent allowed by their commutativity properties) until no more reorderings are possible.

When the optimizer has found all productive transformations using the above tactics, it constructs a new sub-network, binds it into the composite Aurora network that is running, and then instructs the scheduler to stop holding messages at the input connection points. Of course, outputs affected by the sub-network will see a *blip* in response time; however the remainder of the network can proceed unimpeded.

An Aurora network is broken naturally into a collection of k sub-networks by the connection points that are inserted by the application administrator. Each of these sub-networks can be optimized individually, because it is a violation of Aurora semantics to optimize across a connection point. The Aurora optimizer is expected to cycle periodically through all k sub-networks and run as a background task.

3.2 Ad-Hoc Query Optimization

One last issue that must be dealt with is ad-hoc query optimization. Recall that the semantics of an ad-hoc query is that it must run on all the historical information saved at the connection point(s) to which it is connected. Subsequently, it becomes a normal portion of an Aurora network, until it is discarded. Aurora processes ad-hoc queries in two steps by constructing two separate subnetworks. Each is attached to a connection point, so the optimizer can be run before the scheduler lets messages flow through the newly added subnetworks.

Aurora semantics require the historical subnetwork to be run first. Since historical information is organized as a B-tree, the Aurora optimizer begins at each connection point and examines the successor box(es). If the box is a filter, then Aurora examines the condition to see if it is compatible with the storage key associated with the connection point. If so, it switches the implementation of the filter box to perform an indexed lookup in the B-tree. Similarly, if the successor box is a join, then the Aurora optimizer costs performing a merge-sort or indexed lookup, chooses the cheapest one, and changes the join implementation appropriately. Other boxes cannot effectively use the indexed structure, so only these two need be considered. Moreover, once the initial box

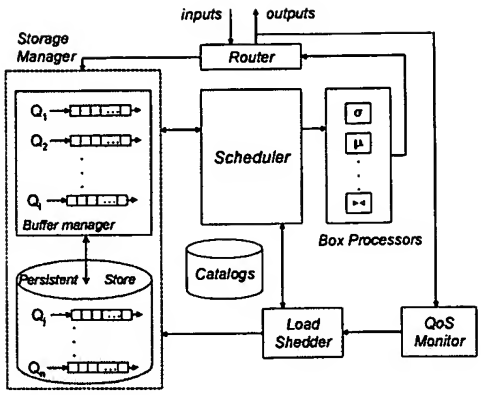


Figure 3: Aurora run-time architecture

performs its work on the historical tuples, the index structure is lost, and all subsequent boxes will work in the normal way. Hence, the optimizer converts the historical subnetwork into an optimized one, which is then executed.

When it is finished, the subnetwork used for continuing operation can be run to produce subsequent output. Since this is merely one of the sub-networks, it can be optimized in the normal way suggested above.

In summary, the initial boxes in an ad-hoc query can *pull* information from the B-tree associated with the corresponding connection point(s). When the historical operation is finished, Aurora switches the implementation to the standard *push-based* data structures, and continues processing in the conventional fashion.

4 Run-Time Operation

The basic purpose of Aurora run-time network is to process data flows through a potentially large workflow diagram. Figure 3 illustrates the basic Aurora architecture. Here, inputs from data sources and outputs from boxes are fed to the router, which forwards them either to external applications or to the storage manager to be placed on the proper queue. The storage manager is responsible for maintaining the box queues and managing the buffer. Conceptually, the scheduler picks a box for execution, ascertains what processing is required, and passes a pointer to the box description (together with a pointer to the box state) to the multi-threaded box processor. The box processor executes the appropriate operation and then forwards the output tuples to the router. The scheduler then ascertains the next processing step and the cycle repeats. The QoS monitor continually monitors system performance and activates the load shedder when it detects an overload situation and poor system performance. The load shedder then sheds load till the performance of the system reaches an acceptable level. The catalog in Figure 3 contains information regarding the network topology, inputs, outputs, QoS information, and relevant statistics (e.g., selectivity, average box processing costs), and is essentially used by all components.

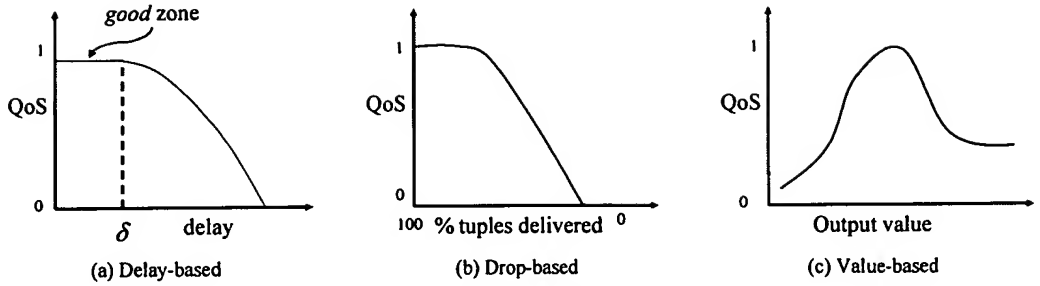


Figure 4: QoS graph types

We now describe Aurora's primary run-time architecture in more detail, focusing primarily on the storage manager, scheduler, QoS monitor, and load shedder.

4.1 QoS Data Structures

Aurora attempts to maximize the perceived QoS for the outputs it produces. QoS, in general, is a multidimensional function of several attributes of an Aurora system. These include:

- *Response times*—output tuples should be produced in a timely fashion; as otherwise QoS will degrade as delays get longer;
- *Tuple drops*—if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate;
- *Values produced*—QoS clearly depends on whether important values are being produced or not.

Asking the application administrator to specify a multidimensional QoS function seems impractical. Instead, Aurora relies on a simpler tactic, which is much easier for humans to deal with: for each output stream, we expect the application administrator to give Aurora a two-dimensional QoS graph based on the processing delay of output tuples produced (as illustrated in Figure 4a). Here, the QoS of the output is maximized if delay is less than the threshold, δ, in the graph. Beyond δ, QoS degrades with additional delay.

Optionally, the application administrator can give Aurora two additional QoS graphs for all outputs in an Aurora system. The first, illustrated in Figure 4b, shows the percentage of tuples delivered. In this case, the application administrator indicates that high QoS is achieved when tuple delivery is near 100% and that QoS degrades as tuples are dropped. The second optional QoS graph for outputs is shown in Figure 4c. The possible values produced as outputs appear on the horizontal axis, and the QoS graph indicates the importance of each one. This value-based QoS graph captures the fact that some outputs are more important than others. For example, in a plant monitoring application, outputs near a critical region are much more important than ones well away from it. Again, if the application administrator has value-based QoS information, then Aurora will use it to shed load more intelligently than would occur otherwise.

Aurora makes several assumptions about the QoS graphs. First, it assumes that all QoS graphs are normalized, so that QoS for different outputs can be quantitatively compared. Second, Aurora assumes that the value chosen for δ is *feasible*, i.e., that a properly sized Aurora network will operate with all outputs in the *good* zone to the left of δ in steady state. This will require the delay introduced by the total computational cost along the longest path from a data source to this output not to exceed δ. If the application administrator does not present Aurora with feasible QoS graphs, then the algorithms in the subsequent sections may not produce good results. Third, unless otherwise stated, Aurora assumes that all its QoS graphs are convex (the value-based graph illustrated in Figure 4c is an exception). This assumption is not only reasonable but also necessary for the applicability of *gradient walking* techniques used by Aurora for scheduling and load shedding.

Note that Aurora's notion of QoS is general and is not restricted to the types of graphs presented here. Aurora can work with other individual attributes (e.g., throughput) or composite attributes (e.g., a weighted, linear combination of throughput and latency) provided that they satisfy the basic assumptions discussed above. In the rest of this paper, however, we restrict our attention to the graph types presented here.

The last item of information required from the application administrator is H, the *headroom* for the system, defined as the percentage of the computing resources that can be used in steady state. The remainder is reserved for the expected ad-hoc queries, which are added dynamically.

4.2 Storage Management

The job of the Aurora Storage Manager (ASM) is to store all tuples required by an Aurora network. There are two kinds of requirements. First, ASM must manage storage for the tuples that are being passed through an Aurora network, and secondly, it must maintain extra tuple storage that may be required at connection points.

Queue Management. Each windowed operation requires a historical collection of tuples to be stored, equal to the size of the window. Moreover, if the network is currently saturated, then additional tuples may accumulate at various places in the network. As such, ASM must manage a collection of variable length queues of tuples. There is one queue at the output of each box, which is shared by all successor boxes. Each such successor box maintains two pointers into this queue. The *head* indicates the oldest tuple that this box has not processed. The *tail*, in contrast, indicates the oldest tuple that the box needs. The head and tail indicate box's current window, which slides

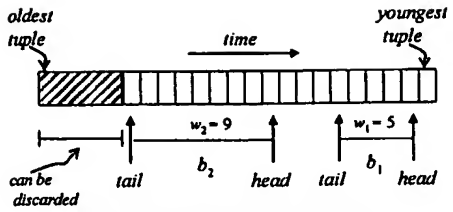


Figure 5: Queue organization

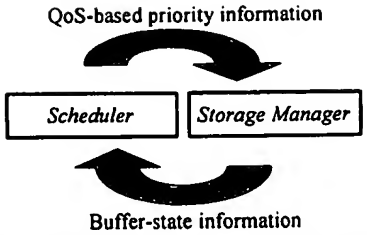


Figure 6: Scheduler-storage manager interaction

as new tuples are processed. ASM will keep track of these collections of pointers, and can normally discard tuples in a queue that are older than the oldest tail pointing into the queue. In summary, when a box produces a new tuple, it is added to the front of the queue. Eventually, all successor boxes process this tuple and it falls out of all of their windows and can be discarded. Figure 5 illustrates this model by depicting a two-way branch scenario where two boxes, b_1 and b_2, share the same queue ('w's refer to window sizes).

Normally, queues of this sort are stored as main memory data structures. However, ASM must be able to scale arbitrarily, and has chosen a different approach. Disk storage is divided into fixed length blocks, of a tunable size, *block_size*. We expect typical environment will use 128KB or larger blocks. Each queue is allocated one block, and queue management proceeds as above. As long as the queue does not overflow, the single block is used as a circular buffer. If an overflow occurs, ASM looks for a collection of two blocks (contiguous if possible), and expands the queue dynamically to 2 × *block_size*. Circular management continues in this larger space. Of course, queue underflow can be treated in an analogous manner.

At start up time, ASM is allocated a buffer pool for queue storage. It pages queue blocks into and out of main memory using a novel replacement policy. The scheduler and ASM share a tabular data structure that contains a row for each box in the network containing the current scheduling priority of the box and the percentage of its queue that is currently in main memory. The scheduler periodically adjusts the priority of each box, while the ASM does likewise for the main memory residency of the queue. This latter piece of information is used by the scheduler for guiding scheduling decisions (see Section 4.3). The data structure also contains a flag to indicate that a box is currently running. Figure 6 illustrates this interaction.

When space is needed for a disk block, ASM evicts the lowest priority main memory resident block. In addition, whenever, ASM discovers a block for a queue that does not correspond to a running block, it will attempt to "upgrade" the block by evicting it in favor of a block for the queue corresponding to a higher priority box. In this way, ASM is continually trying to keep all the required blocks in main memory that correspond to the top priority queues. ASM is also aware of the size of each queue and whether it is contiguous on disk. Using this information, it can schedule multi-block reads and writes and garner added efficiency. Of course, as blocks move through the system and conditions change, the scheduler will adjust the priority of boxes, and ASM will react by adjusting the buffer pool. Naturally, we must be careful to avoid the well-known *hysteresis* effect, whereby ASM and the scheduler start working at cross purposes, and performance degrades sharply.

Connection Point Management. As noted earlier, the Aurora application designer indicates a collection of connection points, to which collections of boxes can be subsequently connected. This satisfies the Aurora requirement to support ad-hoc queries. Associated with each connection point is a history requirement and an optional storage key. The history requirement indicates the amount of historical information that must be retained. Sometimes, the amount of retained history is less than the maximum window size of the successor boxes. In this case, no extra storage need be allocated. The usual case is that additional history is requested.

In this case, ASM will organize the historical tuples in a B-tree organized on the storage key. If one is not specified, then a B-tree will be built on the timestamp field in the tuple. When tuples fall off the end of a queue that is associated with a connection point, then ASM will gather up batches of such tuples and insert them into the corresponding B-tree. Periodically, it will make a pass through the B-tree and delete all the tuples, which are older than the history requirement. Obviously, it is more efficient to process insertions and deletions in batches, than one by one.

Since we expect B-tree blocks to be smaller than *block_size*, we anticipate splitting one or more of the buffer pool blocks into smaller pieces, and paging historical blocks into this space. The scheduler will simply add the boxes corresponding to ad-hoc queries to the data structure mentioned above, and give these new boxes a priority. ASM will react by prefetching index blocks, but not data blocks, for worthy indexed structures. In turn, it will retain index blocks, as long as there are not higher priority buffer requirements. No attempt will be made to retain data blocks in main memory.

4.3 Real-Time Scheduling

Scheduling in Aurora is a complex problem due to the need to simultaneously address several issues including large system scale, real-time performance requirements, and dependencies between box executions. Furthermore, tuple processing in Aurora spans many scheduling and execution

steps (i.e., an input tuple typically needs to go through many boxes before potentially contributing to an output stream) and may involve multiple accesses to secondary storage. Basing scheduling decisions solely on QoS requirements, thereby failing to address end-to-end tuple processing costs, might lead to drastic performance degradation especially under resource constraints. To this end, Aurora not only aims to maximize overall QoS but also makes an explicit attempt to reduce overall tuple execution costs. We now describe how Aurora addresses these two issues.

Train Scheduling. In order to reduce overall processing costs, Aurora observes and exploits two basic *non-linearities* when processing tuples:

- *Inter-box non-linearity*: End-to-end tuple processing costs may drastically increase if buffer space is not sufficient and tuples need to be shuttled back and forth between memory and disk several times throughout their lifetime. One important goal of Aurora scheduling is, thus, to minimize tuple trashing. Another form of inter-box non-linearity occurs when passing tuples between box queues. If the scheduler can decide in advance that, say, box b_2 is going to be scheduled right after box b_1 (whose outputs feed b_2), then the storage manager can be bypassed (assuming there is sufficient buffer space) and its overhead avoided while transferring b_1's outputs to b_2's queue.
- *Intra-box non-linearity*: The cost of tuple processing may decrease as the number of tuples that are available for processing at a given box increases. This reduction in unit tuple processing costs may arise due to two reasons. First, the total number of box calls that need to be made to process a given number of tuples decreases, cutting down low-level overheads such as calls to the box code and context switch. Second, a box, depending on its semantics, may optimize its execution better with larger number of tuples available in its queue. For instance, a box can materialize intermediate results and reuse them in the case of windowed operations, or use merge-join instead of nested loops in the case of joins.

Aurora exploits the benefits of non-linearity in both inter-box and intra-box tuple processing primarily through *train scheduling*, a set of scheduling heuristics that attempt to (1) have boxes queue as many tuples as possible without processing—thereby generating long tuple trains; (2) process complete trains at once—thereby exploiting intra-box non-linearity; and (3) pass them to subsequent boxes without having to go to disk—thereby exploiting inter-box non-linearity. To summarize, train scheduling has two goals: its primary goal is to minimize the number of I/O operations performed per tuple. A secondary goal is to minimize the number of box calls made per tuple.

One important implication of train scheduling is that, unlike traditional blocking operators that wake up and process new input tuples as they arrive, Aurora scheduler tells each box when to execute and how many queued tuples to process. This somewhat complicates the implementation and increases the load of the scheduler, but is necessary for creating and processing tuple trains, which will significantly decrease overall execution costs.

Priority Assignment. The latency of each output tuple is the sum of the tuple's processing delay and its waiting delay. Unlike the processing delay, which is a function of input tuple rates and box costs, the waiting delay is primarily a function of scheduling. Aurora's goal is to assign priorities to outputs so as to achieve the per-output waiting delays that maximize the overall QoS.

The priority of an output is an indication of its urgency. Aurora currently considers two approaches for priority assignment. The first one, a *state-based* approach, assigns priorities to outputs based on their expected utility under the current system state, and then picks for execution, at each scheduling instance, the output with the highest utility. In this approach, the utility of an output can be determined by computing how much QoS will be *sacrificed* if the execution of the output is deferred. A second, *feedback-based* approach continuously observes the performance of the system and dynamically reassigns priorities to outputs, properly increasing the priorities of those that are not doing well and decreasing priorities of the applications that are already in their *good zones*.

Putting It All Together. Because of the large scale, highly dynamic nature of the system, and the granularity of scheduling, searching for optimal scheduling solutions is clearly infeasible. Aurora therefore uses heuristics to simultaneously address real-time requirements and cost reduction by first assigning priorities to select individual outputs and then exploring opportunities for constructing and processing tuple trains.

We now describe one such heuristic used by Aurora. Once an output is selected for execution, Aurora will find the first downstream box whose queue is in memory (note that for a box to be schedulable, its queue must at least contain its window's worth of tuples). Going upstream, Aurora will then consider other boxes, until either it considers a box whose queue is not in memory or it runs out of boxes. At this point, there is a sequence of boxes (i.e., a *superbox*) that can be scheduled one after another.

In order to execute a box, Aurora contacts the storage manager and asks that the queue of the box be pinned to the buffer throughout box's execution. It then passes the location of the input queue to the appropriate box processor code, specifies how many tuples the box should process, and assigns it to an available worker thread.

4.4 Introspection

Aurora employs static and run-time introspection techniques to predict and detect overload situations.

Static Analysis. The goal of static analysis is to determine if the hardware running the Aurora network is sized correctly. If insufficient computational resources are present to handle the steady state requirements of an Aurora network, then queue lengths will increase without bound and response times will become arbitrarily large.

As described before, each box b in an Aurora network has an expected tuple processing cost, $c(b)$, and a

selectivity, $s(b)$. If we also know the expected rate of tuple production $r(d)$ from each data source d, then we can use the following static analysis to ascertain if Aurora is sized correctly.

From each data source, we begin by examining the immediate downstream boxes: if box b_i is directly downstream from data source d_i, then, for the system to be stable, the throughput of b_i should be at least as large as the input data rate; i.e.,

$$1/c(b_i) \geq r(d_i)$$

We can then calculate the output data rate from b_i as:

$$min(1/c(b_i), r(d_i)) \times s(b_i)$$

Proceeding iteratively, we can compute the output data rate and computational requirements for each box in an Aurora network. We can then calculate the minimum aggregate computational resources required per unit time, *min_cap*, for stable steady-state operation. Clearly, the Aurora system with a capacity C cannot handle the expected steady state load if C is smaller than *min_cap*. Furthermore, the response times will assuredly suffer under the expected load of ad-hoc queries if

$$C \times H < min\_cap$$

Clearly, this is an undesirable situation and can be corrected by redesigning applications to change their resource requirements, by supplying more resources to increase system capacity, or by load shedding.

Dynamic Analysis. Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable, long-duration spikes in input rates may deteriorate performance to a level that renders the system useless. We now describe two run-time techniques to detect such cases.

Our technique for detecting an overload relies on the use of delay-based QoS information. Aurora timestamps all tuples from data sources as they arrive. Furthermore, all Aurora operators preserve the tuple timestamps as they produce output tuples (if an operator has multiple input tuples, then the earlier timestamp is preserved). When Aurora delivers an output tuple to an application, it checks the corresponding delay-based QoS graph (Figure 4a) for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone).

4.5 Load Shedding

When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora tuple processing via *load shedding*. The naïve approach to load shedding involves dropping tuples at random points in the network in an entirely uncontrolled manner. This is similar to dropping overflow packets in packet-switching networks [27], and has two potential problems: (1) overall system utility might be degraded more than necessary; and (2) application semantics might be arbitrarily affected. In order to alleviate these problems, Aurora relies on QoS information to guide the load shedding process. We now describe two load-shedding techniques that differ in the way they exploit QoS.

Load Shedding by Dropping Tuples. The first approach addresses the former problem mentioned above: it attempts to minimize the degradation (or maximize the improvement) in the overall system QoS; i.e., the QoS values aggregated over all the outputs. This is accomplished by dropping tuples on network branches that terminate in *more tolerant* outputs.

If load shedding is triggered as a result of static analysis, then we cannot expect to use delay-based or value-based QoS information (without assuming the availability of a priori knowledge of the tuple delays or frequency distribution of values). On the other hand, if load shedding is triggered as a result of dynamic analysis, we can also use delay-based QoS graphs.

We use a greedy algorithm to perform load shedding. Let us initially describe the static load shedding algorithm driven by drop-based QoS graphs. We first identify the output with the *smallest* negative slope for the corresponding QoS graph. We move horizontally along this curve until there is another output whose QoS curve has a smaller negative slope at that point. This horizontal difference gives us an indication of the *output* tuples to drop (i.e., the selectivity of the drop box to be inserted) that would result in the minimum decrease in the overall QoS. We then move the corresponding drop box as far upstream as possible until we find a box that affects other outputs (i.e., a *split point*), and place the drop box at this point. Meanwhile, we can calculate the amount of recovered resources. If the system resources are still not sufficient, then we repeat the process.

For the run-time case, the algorithm is similar except that we can use delay-based QoS graphs to identify the problematic outputs, i.e., the ones that are beyond their delay thresholds, and we repeat the load shedding process until the latency goals are met.

In general, there are two subtleties in dynamic load shedding. First, drop boxes inserted by the load shedder should be among the ones that are given higher priority by the scheduler. Otherwise, load shedding will be ineffective in reducing the load of the system. Therefore, the load shedder simply does not consider the *inactive* (i.e., low priority) outputs, which are indicated by the scheduler. Secondly, the algorithm tries to move the drop boxes as close to the sources as possible to discard tuples before they redundantly consume any resources. On the other hand, if there is a box with a large existing queue, it makes sense to *temporarily* insert the drop box at that point rather than trying to move it upstream closer towards the data sources.

Presumably, the application is coded so that it can tolerate missing tuples from a data source caused by communication failures or other problems. Hence, load shedding simply artificially introduces additional missing tuples. Although the semantics of the application are somewhat different, the harm should not be too damaging.

Semantic Load Shedding by Filtering Tuples. The load shedding scheme described above effectively reduces the amount of Aurora processing by dropping *randomly selected* tuples at strategic points in the network. While this

approach attempts to minimize the loss in overall system utility, it fails to control the impact of the dropped tuples on application semantics. Semantic load shedding addresses this limitation by using value-based QoS information, if available. Specifically, semantic load shedding drops tuples in a more controlled way; i.e., it drops less important tuples, rather than random ones, using filters.

If value-based QoS information is available, then Aurora can watch each output and build up a histogram containing the frequency with which value ranges have been observed. In addition, Aurora can calculate the expected utility of a range of outputs by multiplying the QoS values with the corresponding frequency values for every interval and then summing these values. To shed load, Aurora identifies the output with the *lowest utility interval*; converts this interval to a filter predicate; and then, as before, attempts to propagate the corresponding filter box as far upstream as possible to a split point. This strategy, which we refer to as *backward interval propagation*, admittedly has limited scope because it requires the application of the inverse function for each operator passed upstream (Aurora boxes do not necessarily have inverses). In an alternative strategy, *forward interval propagation*, Aurora starts from an output and goes upstream until it encounters a split point (or reaches the source). It then *estimates* a proper filter predicate and propagates it in downstream direction to see what results at the output. By trial-and-error, Aurora can converge on a desired filter predicate. Note that a combination of these two strategies can also be utilized. First, Aurora can apply backward propagation until a box, say *b*, whose operator's inverse is difficult to compute. Aurora can then apply forward propagation between the insertion location of the filter box and *b*. This algorithm can be applied iteratively until sufficient load is shed.

5 Related Work

A special case of Aurora processing is as a continuous query system. A system like Niagara [7] is concerned with combining multiple data sources in a wide area setting, while we are initially focusing on the construction of a general stream processor that can process very large numbers of streams.

Query indexing [3] is an important technique for enhancing the performance of large-scale filtering applications. In Aurora, this would correspond to a merge of some inputs followed by a fanout to a large number of filter boxes. Query indexing would be useful here, but it represents only one Aurora processing idiom.

As in Aurora, active databases [21, 22] are concerned with monitoring conditions. These conditions can be a result of any arbitrary update on the stored database state. In our setting, updates are append-only, thus requiring different processing strategies for detecting monitored conditions. Triggers evaluate conditions that are either true or false. Our framework is general enough to support queries over streams or the conversion of these queries into monitored conditions. There has also been extensive work on making active databases highly scalable (e.g., [11]). Similar to continuous query research, these efforts have focused on query indexing, while Aurora is constructing a more general system.

Adaptive query processing techniques (e.g., [4, 13, 26]) address efficient query execution in unpredictable and dynamic environments by revising the query execution plan as the characteristics of incoming data changes. Of particular relevance is the Eddies work [4]. Unlike traditional query processing where every tuple from a given data source gets processed in the same way, each tuple processed by an Eddy is dynamically routed to operator threads for partial processing, with the responsibility falling upon the tuple to carry with it its processing state. Recent work [17] extended Eddies to support the processing of queries over streams, mainly by permitting Eddies systems to process multiple queries simultaneously and for unbounded lengths of time. The Aurora architecture bears some similarity to that of Eddies in its division of a single query's processing into multiple threads of control (one per query operator). However, queries processed by Eddies are expected to be processed in their entirety; there is neither the notion of load shedding, nor QoS.

Previous work on stream data query processing architectures shares many of the goals and target application domains with Aurora. The Streams project [5] attempts to provide complete DBMS functionality along with support for continuous queries over streaming data. The Fjords architecture [16] combines querying of push-based sensor sources with pull-based traditional sources by embedding the pull/push semantics into queues between query operators. It is fundamentally different from Aurora in that operator scheduling is governed by a combination of schedulers specific to query threads and operator-queue interactions. Tribeca [25] is an extensible, stream-oriented data processor designed specifically for supporting network traffic analysis. While Tribeca incorporates some of the stream operators and compile-time optimizations Aurora supports, it does not address scheduling or load shedding issues, and does not have the concept of ad-hoc queries.

Work in sequence databases [24] defined sequence definition and manipulation languages over discrete data sequences. The Chronicle data model [14] defined a restricted view definition and manipulation language over append-only sequences. Aurora's algebra extends the capabilities of previous proposals by supporting a wider range of window processing (i.e., Tumble, Slide, Latch), classification (i.e., GroupBy), and interpolation (i.e., Resample) techniques.

Our work is also relevant to materialized views [10], which are essentially stored continuous queries that are re-executed (or incrementally updated) as their base data are modified. However, Aurora's notion of continuous queries differs from materialized views primarily in that Aurora updates are append-only, thus, making it much easier to incrementally materialize the view. Also, query results are streamed (rather than stored); and high stream data rates may require load shedding or other approximate query

processing techniques that trade off efficiency for result accuracy.

Our work is likely to benefit from and contribute to the considerable research on temporal databases [20], main-memory databases [8], and real-time databases [15, 20]. These studies commonly assume an HADP model, whereas Aurora proposes a DAHP model that builds streams as fundamental Aurora objects. In a real-time database system, transactions are assigned timing constraints and the system attempts to ensure a degree of confidence in meeting these timing requirements. The Aurora notion of QoS extends the soft and hard deadlines used in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual transactions, whereas Aurora associates QoS curves with outputs from stream processing and, thus, has to support continuous timing requirements. Relevant research in workflow systems (e.g., [18]) primarily focused on organizing long-running interdependent activities but did not consider real-time processing issues.

There has been extensive research on scheduling tasks in real-time and multimedia systems and databases [19, 20]. The proposed approaches are commonly deadline driven; i.e., at each scheduling point, the task that has the earliest deadline or one that is expected to provide the highest QoS (e.g., throughput) is identified and scheduled. In Aurora, such an approach is not only impractical because of the sheer number of potentially schedulable tasks (i.e., tuples), but is also inefficient because of the implicit assumption that all tasks are memory-resident and are scheduled and executed in their entirety. To the best of our knowledge, however, our train scheduling approach is unique in its ability to reduce overall execution costs by exploiting intra- and inter-box non-linearities described here.

The work of [26] takes a scheduling-based approach to query processing; however, they do not address continuous queries, are primarily concerned with data rates that are too slow (we also consider rates that are too high), and they only address query plans that are trees with single outputs.

The congestion control problem in data networks [27] is relevant to Aurora and its load shedding mechanism. Load shedding in networks typically involves dropping individual packets randomly, based on timestamps, or using (application-specified) priority bits. Despite conceptual similarities, there are also some fundamental differences between network load shedding and Aurora load shedding. First, unlike network load shedding which is inherently distributed, Aurora is aware of the entire system state and can potentially make more intelligent shedding decisions. Second, Aurora uses QoS information provided by the external applications to trigger and guide load shedding. Third, Aurora's semantic load shedding approach not only attempts to minimize the degradation in overall system utility, but also quantifies the imprecision due to dropped tuples.

Aurora load shedding is also related to approximate query answering (e.g., [12]), data reduction, and summary techniques [6, 9], where result accuracy is traded for efficiency. By throwing away data, Aurora bases its computations on sampled data, effectively producing approximate answers using data sampling. The unique aspect of our approach is that our sampling is driven by QoS specifications.

6 Implementation Status

As of June 2002, we have a prototype Aurora implementation. The prototype has a Java-based GUI that allows construction and execution of Aurora networks. The interface is currently primitive, but will be extended over the next few months to support specification of QoS graphs, connection points, and zoom. The run-time system contains a primitive scheduler, a rudimentary storage manager, and code to execute most of the boxes. Aurora metadata is stored in a schema, which is stored in a Berkeley DB [1] database. Hence, Aurora is functionally complete, and multi-box networks can be constructed and run. However, there is currently no optimizer and load shedding. We expect to implement Aurora functionality in these areas over the course of the summer.

7 Conclusions and Future Work

Monitoring applications are those where streams of information, triggers, real-time requirements, and imprecise data are prevalent. Traditional DBMSs are based on the HADP model, and thus cannot provide adequate support for such applications. In this paper, we have described the architecture of Aurora, a DAHP system oriented towards monitoring applications. We argued that providing efficient support for these demanding applications not only require critically revisiting many existing aspects of database design and implementation, but also require developing novel proactive data storage and processing concepts and techniques.

In this paper, we first presented the basic Aurora architecture, along with the primitive building blocks for workflow processing. We followed with several heuristics for optimizing a large Aurora network. We then focused on run-time data storage and processing issues, discussing storage organization, real-time scheduling, introspection, and load shedding, and proposed novel solutions in all these areas.

We are currently implementing an Aurora prototype system, which we will use to investigate the practicality and efficiency of our proposed solutions. We are also investigating two important research directions. While the bulk of the discussion in this paper describes how Aurora works on a single computer, many stream-based applications demand support for distributed processing. To this end, we are working on a distributed architecture, Aurora*, which will enable operators to be pushed closer to the data sources, potentially yielding significantly improved scalability, energy use, and bandwidth efficiency. Aurora* will provide support for distribution by running a full Aurora system on each of a collection of communicating nodes. In particular, Aurora* will manage load by replicating boxes along a path and migrating a copy

of this sub-network to another more lightly loaded node. A subset of the stream inputs to the replicated network would move along with the copy. We are also extending our basic data and processing model to cope with missing and imprecise data values, which are common in applications involving sensor-generated data streams.

References

[1] Berkeley DB. Sleepycat Software, http://www.sleepycat.com/.

[2] Informix White Paper. Time Series: The Next Step for Telecommunications Data Management.

[3] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, Cairo, Egypt, 2000.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, Dallas, TX, 2000.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109-120, 2001.

[6] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4):3-45, 1997.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, Dallas, TX, 2000.

[8] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509-516, 1992.

[9] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, Santa Barbara, CA, 2001.

[10] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[11] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proc. of the 15th Intl. Conf. on Data Engineering*, Sydney, Austrialia, 1999.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, Tucson, 1997.

[13] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, Philadelphia, PA, 1999.

[14] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proc. of the 14th Symposium on Principles of Database Systems*, San Jose, CA, 1995.

[15] B. Kao and H. Garcia-Molina. An Overview of Realtime Database Systems. In *Real Time Computing*, W. A. Halang and A. D. Stoyenko, Eds.: Springer-Verlag, 1994.

[16] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proc. of the 18th Intl. Conf. on Data Engineering*, San Jose, CA, 2002.

[17] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, Wisconsin, USA, 2002.

[18] C. Mohan, D. Agrawal, G. Alonso, A. E. Abbadi, R. Gunther, and M. Kamath. Exotica: A Project on Advanced Transaction Management and Workflow Systems. *SIGOIS Bulletin*, 16(1):45-50, 1995.

[19] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. of the 16th Intl. ACM Symposium on Operating Systems Principles*, 1997.

[20] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513-532, 1995.

[21] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63-103, 1999.

[22] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases*, Barcelona, Spain, 1991.

[23] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262-266, 1990.

[24] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proc. of the 22nd Intl. Conf. on Very Large Data Bases*, Bombay, India, 1996.

[25] M. Sullivan and A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proc. of the USENIX Annual Technical Conf.*, New Orleans, LA, 1998.

[26] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, Rome, Italy, 2001.

[27] C. Yang and A. V. S. Reddy. A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. *IEEE Network*, 9(5):34-44, 1995.

On Computing Correlated Aggregates Over Continual Data Streams*

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Flip Korn
AT&T Labs–Research
flip@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

In many applications from telephone fraud detection to network management, data arrives in a stream, and there is a need to maintain a variety of statistical summary information about a large number of customers in an online fashion. At present, such applications maintain basic aggregates such as running extrema values (`MIN`, `MAX`), averages, standard deviations, etc., that can be computed over data streams with limited space in a straightforward way. However, many applications require knowledge of more complex aggregates relating different attributes, so-called *correlated aggregates*. As an example, one might be interested in computing the percentage of international phone calls that are longer than the average duration of a domestic phone call. Exact computation of this aggregate requires multiple passes over the data stream, which is infeasible.

We propose single-pass techniques for approximate computation of correlated aggregates over both landmark and sliding window views of a data stream of tuples, using a very limited amount of space. We consider both the case where the independent aggregate (average duration in the example above) is an extrema value and the case where it is an average value, with any standard aggregate as the dependent aggregate; these can be used as building blocks for more sophisticated aggregates. We present an extensive experimental study based on some real and a wide variety of synthetic data sets to demonstrate the accuracy of our techniques. We show that this effectiveness is explained by the fact that our techniques exploit monotonicity and convergence properties of aggregates over data streams.

1. INTRODUCTION

In many applications from telephone fraud detection to network management, data arrives in a *stream*, and online decisions are made based on a "recently observed" portion of the stream. For example, telephone call records are generated for each call, at the end of the call. The stream of generated call records may be used by applications such as telephone fraud detection, which examines this stream for various patterns of potentially fraudulent calling behavior. As another example, router interfaces are periodically polled by network operators using SNMP to get a variety of performance data. The stream of polled SNMP data is then used by network monitoring and management applications to determine if an interface is down, router is inaccessible, *etc.*

The large volume of stream data (hundreds of millions of phone call records from tens of millions of customers per day; tens of millions of SNMP records from tens of thousands of router interfaces per day, *etc.*), and the online nature of the various applications that operate on such data, makes it imperative for the applications to compute and maintain a variety of statistical summary information in an online fashion. At present, such applications maintain basic aggregates such as running averages, standard deviations, *etc.*, that can be computed over data streams with limited storage in a straightforward way. However, more complex aggregates are often desired, especially when exploring correlations between attributes of tuples in the data stream; this application scenario allows users to specify *ad hoc* complex aggregates as the data stream flows by, and to request that results be computed and reported periodically. For example, for each telephone customer, what percentage of calls longer than the average duration are international calls? Or, for each router interface, how often is the total outbound traffic within, say, 50% of the maximum outbound traffic?

Correlated aggregates [6, 5, 4] provide a natural mechanism for the flexible composition of standard aggregates, that are useful for such applications. For example, $\texttt{COUNT}\{x : x > 0.5*\texttt{MAX}(x)\}$ operates on a multiset of x values, and computes the number of x values that are within 50% of the maximum x value in the multiset. Similarly, $\texttt{MAX}\{y : x < \texttt{AVG}(x)\}$ operates on a multiset of (x, y) tuples, and computes the maximum y value obtained from tuples where the x value is less than the average x value in the set; this value may not be the maximum y value in the entire multiset. Prior work [6, 5, 4] has considered only the exact computation of correlated aggregates over finite data sets. In general, this exact computation requires multiple passes over the data set. The first pass is needed to determine the independent aggregate (average duration of calls, or the maximum outbound traffic, in our applications). The second pass is needed to determine the dependent aggregate (percentage of calls longer than the

*Work of Johannes Gehrke supported in part by a gift from Microsoft Corporation and an IBM Faculty Development Award.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

(previously computed) value, or the number of times the total traffic is within 50% of the (previously computed) value. For data streams of large volume, such exact computation is not feasible, and providing a quick approximate answer will have to suffice. The problem of *efficient approximate computation of correlated aggregates over data streams* is the focus of this paper, and we make the following main contributions:

- We classify correlated aggregates over data streams, based on their *scope* and the nature of the *independent aggregate*.

  We illustrate via examples that two natural alternatives for the scope are *landmark windows*, and *sliding windows*. Landmark windows identify certain landmarks in the data stream, and the aggregate value at a point is defined with respect to the tuples from the immediately-preceding landmark until the current point; for example, correlated aggregates on a daily basis, or from the point where a user requested the computation of an *ad hoc* aggregate. Sliding windows are typically of a fixed width, and the aggregate value at a point is defined with respect to the tuples that precede this point and are within this width; for example, correlated aggregates over the previous 60 minutes.

  We consider both the case where the independent aggregate is an extrema value (`MIN`, `MAX`) and the case where it is an average value; any standard aggregate can be used as a dependent aggregate.

- Since correlated aggregates typically cannot be computed exactly in a single pass (equivalently, the exact computation of correlated aggregates in a single pass requires an unbounded buffer size), it is natural to expect that the accuracy of a single-pass, approximate computation of correlated aggregates directly depends on the additional "buffer" space available for maintaining an estimate of the aggregate value.

  Histograms have been used widely both in the research literature and in commercial DBMSs for quickly estimating approximate statistics about underlying data domains, in limited space. Hence, the use of traditional histograms, with few buckets (depending on the buffer space available per aggregate), is a viable strawman for our problem. We observed two limitations of the use of standard histograms with respect to our problem:

  - First, at a point during the stream computation, we may be interested only in a focused small, subinterval of the entire range of values from the underlying domain. Since traditional histogram techniques (equiwidth, equidepth, *etc.*) allocate buckets over the *entire range of values in the domain*, they often waste buckets allocating them to regions that cannot (or are unlikely to) affect subsequent results.
  - Second, the region of interest (*e.g.*, the tuples whose x values are less than $\texttt{AVG}(x)$) may shift while the data stream is passing by. Traditional work on histograms has not addressed this issue, to the best of our knowledge.

  These two issues motivate us to seek novel solutions to the problem of accurately maintaining histogram information over a stream of data records, which can effectively deal with a dynamically changing (shifting, contracting, expanding) region of interest.

- We propose single-pass techniques for the approximate computation of correlated aggregates that adapt the histogram bucket boundaries to a dynamically changing region of interest in one of two ways, over both landmark and sliding window data streams of tuples: (i) a *wholesale* approach that can change each and every bucket boundary in response to a change in the region of interest; and (ii) a *piecemeal* approach that is more conservative and changes bucket boundaries only when absolutely necessary.

  We show how properties of the independent aggregate (monotonicity of extrema, and convergence of average) can be used effectively for the approximate computation of correlated aggregates, over landmark window data streams.

  Over sliding windows, our techniques need to deal with the simultaneous inclusion of a data tuple and exclusion of another data tuple, in efficiently computing the correlated aggregate. We achieve this by combining the approach we use for cumulative data streams with novel incremental mechanisms for approximately maintaining extrema aggregates in a sliding window.

- We complement our algorithmic analysis with an extensive experimental study based on some real and some synthetic data sets to demonstrate the accuracy of our techniques for approximate computation of correlated aggregates. The main conclusions of our study are as follows:

  - Use of focused histograms to estimate correlated aggregates over data streams is demonstrably superior, in a large variety of cases, to the use of traditional histograms.
  - A simple "piecemeal" strategy, which maintains uniformly-spaced bucket boundaries in its region of interest, and changes bucket boundaries only at the extremities of the region, when the region of interest changes, is the strategy of choice across a wide variation in window scopes, nature of aggregate, and type of data stream.

The principal consequence of our study is that we have a robust approach for the approximate computation of the important class of correlated aggregates over streaming data. To the best of our knowledge, ours is the first work on evaluating this important class of aggregates over streaming data.

2. PROBLEM DEFINITION

2.1 Data Streams

Consider a relational schema R with attributes $X_1, \ldots, X_k$ where attribute X_i has $\texttt{dom}(X_i)$. We call $\texttt{att}(R) \stackrel{\text{def}}{=} \texttt{dom}(R) \stackrel{\text{def}}{=} \texttt{dom}(X_1) \times \cdots \times \texttt{dom}(X_k)$ the *attribute space of* R. Let R be a relational schema with attribute space $\texttt{att}(R)$. We call

a function $O : \mathbf{N} \to \mathtt{att}(R)$ an *ordering of* R. A *sequence* is a tuple $S(R, O)$ where R is a relational schema and O is an ordering of R. Given a sequence $S(R_S, O_S)$, we also refer to the natural numbers as *positions*, and we use $S[i]$ for $O_S(i)$, the record at the ith position.

Our model of computation is similar to the model introduced by Henzinger, Raghavan, and Rajagopalan [19]. It contains a single input sequence S_{in} and a single output sequence S_{out}, and the model has as single parameter m, the amount of space available. Computation in our model proceeds in steps, and each computation step consists of three substeps. Consider the ith computation step. In the first substep, we read $S_{in}[i]$ from the input sequence S_{in} into a memory location; in the second substep, we perform an unlimited amount of computation in memory; and in the third substep we write into the ith position of the output sequence $S_{out}[i]$. We call an algorithm for our model of computation a *stream algorithm*. Thus, our algorithms map input streams into output streams, which could be used for further processing.

We consider stream algorithms for aggregate computations. A *stream aggregate* has three components: A *scalar aggregate function* $AGG : 2^{\mathbf{R}} \to \mathbf{R}$, a *scope function* $scope : \mathbf{N} \to 2^{\mathbf{N}}$, and a *selection predicate* P. Given an input sequence S_{in}, a stream aggregate operator $Agg(AGG, scope, P)$ returns the sequence S_{out} such that

$$S_{out}[i] = AGG\{S_{in}[j].X_l \mid j \in scope(i) \wedge P(S_{in}[j], S_{in}[scope(i)])\}$$

where $S_{in}[scope(i)] \stackrel{\text{def}}{=} \{S_{in}[j] \mid j \in scope(i)\}$ and X_l is an attribute of R.

Three particular types of scope functions are especially interesting. We call the scope function $fScope : \mathbf{N} \to 2^{\mathbf{N}}$ such that $fScope(i) = \{1, \ldots, i\}$ for $i \in \mathbf{N}$ a *full window scope*, and we call the scope function $swScope_w$ such that $swScope_w(i) = \{\max(1, i-w+1), \max(1, i-w+2), \ldots, i\}$ a *sliding window scope of size* w. A full window scope is just a special case of a *landmark window scope*. A landmark window scope takes as input a *landmark set* $S = \{s_1, s_2, \ldots\}$. Given such a set S and a position i, $lmScope(S, i) = \{s_j, s_j+1, \ldots, i-1, i\}$, where s_j is the largest position in S that is $\leq i$ where S is understood from the context. We omitted and simply use $lmScope(i)$. In this paper, we concentrate on sliding window scopes and on landmark window scopes.

Consider the stream aggregate operator $Agg(AGG, scope, P)$. If the selection predicate P does not contain any aggregate function, then we call Agg a *level* 0 stream aggregate operator. Recursively, let Agg be a level i stream aggregate operator. Then $Agg'(AGG', scope', P')$ is a level $i+1$ stream aggregate operator if

$$S_{out}[i] = AGG'\{S_{in}[j].X_l \mid j \in scope'(i) \wedge P'(S_{in}[j], Agg(S_{in})[i])\}$$

Our notion of the level of a stream aggregate lets us relate stream aggregates to regular queries over a static relation. A level i stream aggregate can be evaluated over a static relation in at most $i+1$ scans. Note that our notation of a level i stream aggregate is purely syntactic. There are level i stream aggregates that have equivalent level 0 aggregates for any i. Establishing such equivalences is outside the scope of this paper.

Consider the following application scenario from the telecommunications industry. Our data stream contains information about phone calls, captured in the following schema:

`CallDetail (origin, dialed, time, duration, isIntl)`

The attributes `origin` and `dialed` contain the originating and destination phone number of the call, respectively; the attributes `time` and `duration` denote the start time and duration of the phone call, respectively; the attribute `isIntl` indicates whether the call was an international phone call. We are interested in computing the following stream aggregates.

Example 1: At any point in time, we would like to compute the number of international calls over the last two months that took longer than 10 minutes. This is an example of a level 0 stream aggregate with a window scope of two months. With respect to our notation, this stream aggregate can be expressed as follows:

$$S_{out}[i] = \mathtt{COUNT}\{S_{in}[j].\mathtt{origin} \mid j \in swScope(i) \wedge S_{in}[j].\mathtt{isIntl} = 1 \wedge S_{in}[j].\mathtt{duration} > 10\},$$

where $swScope$ is the appropriate sliding window scope that restricts relevant records from the input sequence S_{in} to the last two months. □

Example 2: At any point in time, we would like to find the number of international calls this year that were longer than the average call duration. Note that if `CallDetail` were a regular relation, then evaluation of this query would require two passes over the (materialized) relation. In the first pass, we would compute the average call duration d, and then in the second pass we would compute the number of international calls that have a duration longer than d. This is an example of a level 1 stream aggregate with a landmark window scope; the landmark set consists of the beginnings of each year. With respect to our notation, this stream aggregate can be expressed as follows:

$$S_{out}[i] = \mathtt{COUNT}\{S_{in}[j].\mathtt{origin} \mid j \in lmScope(i) \wedge S_{in}[j].\mathtt{isIntl} = 1 \wedge S_{in}[j].\mathtt{duration} \geq \mathtt{AVG}\{S_{in}[k].\mathtt{duration} \mid k \in lmScope(i)\}\},$$

where $lmScope$ is the appropriate landmark window scope that restricts relevant records from the input sequence S_{in} to the current year. □

Example 3: At any point in time, we would like to find the number of international calls whose duration was within 10% of the call with the longest duration with respect to the last two weeks. This is an example of a level 1 stream aggregate with a window scope of two weeks. With respect to our notation, this stream aggregate can be expressed as follows:

$$S_{out}[i] = \mathtt{COUNT}\{S_{in}[j].\mathtt{origin} \mid j \in swScope(i) \wedge S_{in}[j].\mathtt{isIntl} = 1 \wedge S_{in}[j].\mathtt{duration} \geq 0.9 \cdot \mathtt{MAX}\{S_{in}[k].\mathtt{duration} \mid k \in swScope(i)\}\},$$

where $swScope$ is the appropriate sliding window scope that restricts relevant records from the input sequence S_{in} to the last two weeks. □

If the amount of available space m is infinite, then we can compute S_{out} exactly for any stream aggregate through the

following simple algorithm: At step i, we read $S_{in}[i]$ into memory location $M[i]$. We then compute the exact value of the stream aggregate $S_{out}[i]$ using the copy of the input stream being stored and store the output in $S_{out}[i]$. The focus of this paper is on algorithms for computing stream aggregates with a given *constant* amount of space.[1]

In this paper, we focus on level 1 stream aggregates for a relational schema $R(X, Y)$ with two numerical attributes. Specifically, we consider stream aggregates of the form:

$$S_{out}[i] = AGG\text{-}D\{S_{in}[j].Y \mid j \in scope(i) \wedge P(S_{in}[j].X, AGG\text{-}I\{S_{in}[k].X \mid k \in scope(i)\})\},$$

where *AGG-D* and *AGG-I* are aggregate functions, *scope* is a scope function, and P is a simple boolean selection predicate. We call *AGG-D* the *dependent aggregate*, and we call *AGG-I* the *independent aggregate*.

As concrete instantiations, we concentrate in this paper on the following three types of stream aggregates:

- The independent aggregate is either `MIN` or `MAX`. An example instantiation of a stream aggregate operator produces records of the output sequence S_{out} that are computed as follows:

$$S_{out}[i] = AGG\text{-}D\{S_{in}[j].Y \mid j \in scope(i) \wedge \texttt{MIN}\{S_{in}[k].X \mid k \in scope(i)\} \leq S_{in}[j].X \leq (1+\epsilon) \cdot \texttt{MIN}\{S_{in}[k].X \mid k \in scope(i)\}\}.$$

- The independent aggregate is `AVG`. An example instantiation of a stream aggregate operator produces records of the output sequence S_{out} that are computed as follows:

$$S_{out}[i] = AGG\text{-}D\{S_{in}[j].Y \mid j \in scope(i) \wedge S_{in}[j].X \geq \texttt{AVG}\{S_{in}[k].X \mid k \in scope(i)\}\}.$$

2.2 Properties of Aggregate Functions

Let us introduce some properties of stream aggregate operators. These properties motivate our algorithms in Section 3. We call a stream aggregate $Agg(AGG, scope, P)$ *monotonic* if

$$\forall i \in \mathbf{N} : S_{out}[i+1] \leq S_{out}[i] \ \vee \ \forall i \in \mathbf{N} : S_{out}[i+1] \geq S_{out}[i]$$

We will see later that some stream aggregates under landmark scope with `MIN` or `MAX` as the independent aggregate are monotonic. However, this is not the case when `AVG` is the independent aggregate. In this case, we can give tight confidence bounds on the difference between the currently computed value of the aggregate and the (theoretical) expected value μ of the underlying distribution, depending on the step i. Define

$$\widehat{\mu}_i \stackrel{\text{def}}{=} \frac{1}{i}\sum_{j=1}^{i} S_{in}[j], \text{ and } \widehat{\sigma}_i^2 = \frac{1}{i}\sum_{j=1}^{i}(S_{in}[j] - \widehat{\mu}_i)^2$$

[1]If we assume that our algorithms use only a constant amount of space, we neglect here the logarithmic growth of the number of bits that results from computations on very long input sequences. Such sequences require our algorithms to run for many steps and potentially require the storage of very large numbers that grow beyond the precision possible by 32 or 64 bit architecture. We will disregard this logarithmic growth in our space considerations since we do not believe that it is important in practice.

After the ith step, the running value of the average is given by $\widehat{\mu}_i$. Then, by the standard central limit theorem for i.i.d. random variables,

$$\frac{\sqrt{i}(\widehat{\mu}_i - \mu)}{\widehat{\sigma}_i} \Rightarrow N(0,1),$$

as $i \to \infty$, where $\Rightarrow$ denotes convergence in distribution, and $N(0,1)$ denotes a random variable with normal distribution, mean 0 and variance 1. Thus for large values of i, we know that for $\epsilon > 0$

$$P(|\widehat{\mu}_i - \mu| \leq \epsilon) \approx 2 \cdot \Phi(\frac{\epsilon\sqrt{i}}{\widehat{\sigma}_i}) - 1,$$

where Φ is the standardized normal probability distribution.

2.3 Quality Measures

Since our focus is on complex stream aggregates that cannot be computed exactly with a constant amount of space, we need to quantitatively differentiate between our approximation strategies proposed in Sections 3 and 4. Both for landmark scope and for sliding window scope, we would like our algorithms to approximate the exact value as well as possible at every position i of the stream. Let $Agg(AGG, scope, P)$ be a stream aggregate. Define the value of the stream of exact answers S_{exact} at step i as follows:

$$S_{exact}[i] \stackrel{\text{def}}{=} AGG\{S_{in}[j].X_l \mid j \in scope(i) \wedge P(S_{in}, j, scope)\}.$$

Given the exact answer of the aggregate computation, we can define the root mean-squared error $RMSE_n$ at step n as follows:

$$RMSE_n \stackrel{\text{def}}{=} \sqrt{\frac{1}{|scope(i)|}\sum_{j \in scope(i)}(S_{out}[j] - S_{exact}[j])^2}.$$

3. LANDMARK WINDOW ALGORITHMS

In this section we describe algorithms that maintain correlated aggregates for data streams over a landmark window. Our algorithms exploit the monotonicity and convergence properties discussed in Section 2. We then present some experimental results on real and synthetic data streams to validate the effectiveness of our algorithms.

3.1 Description

Here we propose algorithms for maintaining correlated aggregates over landmark-based windows, where the independent aggregate is either an extrema or `AVG`. Our focus is on (one-sided) correlations such as $\texttt{COUNT}\{y : x < (1+\epsilon) * \texttt{MIN}(x)\}$ and $\texttt{COUNT}\{y : x > \texttt{AVG}(x)\}$, although it is straightforward how to extend our techniques to deal with two-sided correlations such as $\texttt{COUNT}\{y : (\texttt{AVG}(x) - \epsilon) < x < (\texttt{AVG}(x) + \epsilon)\}$. For exposition, we describe the case where `COUNT` is used as the dependent aggregate; our techniques easily extend to the case where `SUM` is the dependent aggregate.

Our solution involves the use of histograms as a summary data structure for the incoming data tuples $S_{in}[i]$. The histograms contain an ordered set of m adjacent buckets ordered by abscissae over the x-axis, and are of the form $\langle (v_1, f_1), (v_2, f_2), \ldots, (v_m, f_m) \rangle$. Aggregates are approximated from the histograms in a straightforward way, by estimating the overlap with the existing buckets. The

approximation error in correlated aggregates comes from the truncation of a single bucket (the bucket containing $(1+\epsilon) * \texttt{MIN}(x)$; the bucket containing the mean for AVG). Some assumption (*e.g.*, local uniformity) is required to estimate the number of data tuples in a subrange of a single bucket; however, upper- or lower-bounds can be reported based on counting or discarding the entire bucket, respectively. It is beneficial to allocate most of the space to buckets in the region where approximation error is expected. We exploit the properties described in the previous section to design appropriate strategies to do this. In particular, we exploit the monotonicity of extremas when MIN or MAX is the independent aggregate, and the Central Limit Theorem when AVG is the independent aggregate.

At any given moment, our histogram buckets are tuned for up-to-the-moment data tuples from the stream. However, an existing bucket allocation can degrade from the given partitioning policy (uniform, quantiled) with the arrival of new tuples. We describe two bucket reallocation strategies to compensate for this: *wholesale* and *piecemeal*. The wholesale approach completely revises the set of buckets from scratch at each step based on a new partitioning; the reallocation requires the use of some form of interpolation (*e.g.*, based on uniformity). The piecemeal approach tries to preserve the existing bucket infrastructure while staying consistent with the given bucketing policy, to reduce the approximation error resulting from repeated application of interpolation based on the uniformity assumption. For each reallocation approach, we discuss two bucket partitioning policies: the first partitions buckets uniformly, whereas the second tries to maintain quantiles.

3.1.1 *Sketch of Our Algorithms*

Our algorithms follow the outline given in Figure 1. The subroutines InitializeHistogram and ReallocateHistogram differ based on which independent aggregate is supplied. Also, the conditions for which they are applied are different. First, we describe what these subroutines are for extrema; then we describe what they are for AVG.

Landmark Window Algorithm:
Input: data stream and aggregate query
Output: result stream
  $H \leftarrow$ InitializeHistogram(m);
  for $i = 1$ to w do
    read tuple $S_{in}[i]$;
    if (*condition*$_1$) then $H \leftarrow$ InitializeHistogram(m);
    if (*condition*$_2$) then ReallocateHistogram(H);
    add tuple to appropriate histogram bucket;

Figure 1: Landmark Window Algorithm

3.1.2 *Extrema as the Independent Aggregate*

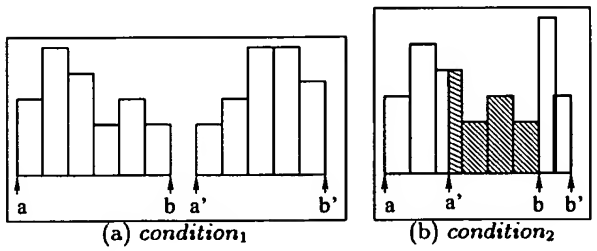
Let $[a, b]$ represent the running range predicate, *e.g.*, $[a, b] = [\min, (1+\epsilon) * \min]$. Given a new minima (or maxima), the range shifts to $[a', b']$. InitializeHistogram is invoked when *condition*$_1$ is satisfied, which occurs when $(b' \leq a)$ for the case of MIN, and $(a' \geq b)$ for the case of MAX. When this occurs, we can reinitialize the histogram and toss out all the tuples seen up to that point without incurring any approximation error. ReallocateHistogram is invoked when *condition*$_2$ is satisfied, which occurs when $(a' \neq a)$ or $(b' \neq b)$. When this occurs, we can truncate the histogram; the resulting approximation error is not cumulative. Figure 2 illustrates the conditions for MAX.

InitializeHistogram: Given the first m tuples $S_{in}[i]$ that arrive, we determine the range $[a, b]$ from the $S_{in}[i].X$-values. However, some of these tuples may be outside this range, so we can purge them from memory. We continue to read tuples, monotonically shifting the range $[a, b]$ based on the arriving x-values, until exactly m tuples remain after the purges.

PartitionHistogram: Uniform partitioning is straightforward: let $v_j = a + j * \frac{b-a}{m}$.

ReallocateHistogram: There are two general approaches for reallocating buckets. In the wholesale approach, we distribute frequences from the old histogram partitioning to the new one; each new frequency is determined by a weighted linear combination of old frequencies. Figure 3(a) gives the pseudocode. In the piecemeal approach, when a new range $[a', b']$ causes some buckets to be truncated, we reallocate the space for new buckets in a manner that best preserves the partitioning policy. Figure 3(b) gives the pseudocode.

Quantiles: In the InitializeHistogram step, we simply sort the first m incoming tuples by x-value. In PartitionHistogram, to get quantiles we start with (v_j, f_j) and determine $(v'_j, \overline{f})$, where $\overline{f} = \frac{1}{m}\sum_{j=1}^{m} f_j$ based on local uniformity assumptions. In the piecemeal approach, buckets can quickly become unbalanced. Our strategy for preserving the quantiles involves occasional merging and splitting of buckets, and is similar to the techniques used in [14]. We periodically check to see when two adjacent buckets can be merged at the same time as a single bucket split. If there is a net gain in improvement, then we perform this merge-split "swap". A merge operation takes two adjacent buckets, (v_j, f_j) and (v_{j+1}, f_{j+1}), and creates a new bucket $(v_j, f_j + f_{j+1})$; a split operation takes a bucket (v_j, f_j) and creates two buckets $(v_j, \frac{f_j}{2})$ and $(v_{j+1}, \frac{f_j}{2})$. We use the variance of the frequencies, which is the standard measure of "goodness" for a quantiled partitioning [14]: $Var(H) = \frac{1}{m}\sum_j (f_j - \overline{f})^2$ where $\overline{f} = \frac{1}{m}\sum f_j$.



(a) *condition*$_1$ (b) *condition*$_2$

Figure 2: Conditions when (a) InitializeHistogram and (b) ReallocateHistogram are called.

3.1.3 AVG *as the Independent Aggregate*

Assume that currently the query range is $[\widehat{\mu}_n, \max]$, where

WholesaleReallocate:
Input: old histogram $H_{old} = \{(v_j, f_j)\}$ in $[a, b]$
Output: new histogram $H_{new} = \{(v'_k, f'_k)\}$ in $[a', b']$
$H_{new} \leftarrow$ PartitionHistogram(H_{old});
if $(a' < a)$ then
  while $(v'_k < v_j)$ $k++$;
else
  while $(v_j < v'_k)$ $j++$;
while $(j < m)$ and $(k < m)$ do
  $f'_k += f_j * \frac{\min\{v_{j+1}, v_{k+1}\} - \max\{v_j, v_k\}}{v_{j+1} - v_j}$
  if $(v'_{k+1} > v_{j+1})$ $j++$;
  else $k++$;

PiecemealReallocate:
Input: old histogram $H_{old} = (v_j, f_j)$ in $[a, b]$
Output: new histogram $H_{new} = (v'_k, f'_k)$ in $[a', b']$
$k \leftarrow j$ such that $b' \in [v_j, v_{j+1}]$;
$v'_k = b'$;
$f_k* = \frac{b - v_k}{v_{k+1} - v_k}$;
keep buckets $\langle (v_1, f_1), ..., (v_k, f_k) \rangle$;
split remaining buckets according to max widths;

Figure 3: Wholesale and Piecemeal Algorithms

$\widehat{\mu}_n = \frac{1}{n}\sum_i^n S_{in}[i].X$ is the running mean from tuples $S_{in}[i]$ that have arrived up to step n. The arrival of a new tuple may cause $\widehat{\mu}_{n+1}$ to shift from $\widehat{\mu}_n$. However, the Central Limit Theorem gives us some indication of the behavior of $\widehat{\mu}_{n+1}$. In particular, it tells us to expect that $\widehat{\mu}_{n+1}$ will remain within the range $[\widehat{\mu}_n - \frac{\widehat{\sigma}_n}{\sqrt{n}}, \widehat{\mu}_n + \frac{\widehat{\sigma}_n}{\sqrt{n}}]$ with 68% probability.[2] Thus, we keep histogram buckets at $(\min, (\widehat{\mu}_n - \frac{\widehat{\sigma}_n}{\sqrt{n}}), \ldots, (\widehat{\mu}_n + \frac{\widehat{\sigma}_n}{\sqrt{n}}), \max)$ where the bucket locations in between are determined by the partitioning policy. InitializeHistogram is invoked once initially, but $condition_1$ is null. ReallocateHistogram is invoked when $condition_2$ is satisfied, which occurs whenever the mean shifts.

InitializeHistogram: Read the first m tuples $S_{in}[i]$ and compute $\widehat{\mu}_n = \frac{1}{m}\sum_i^m S_{in}[i].X$.

PartitionHistogram: We consider two strategies: one that partitions the subinterval $[\widehat{\mu}_n - \frac{\widehat{\sigma}_n}{\sqrt{n}}, \widehat{\mu}_n + \frac{\widehat{\sigma}_n}{\sqrt{n}}]$ uniformly, and one that partitions the interval according to the quantiles of the normal distribution with mean $\widehat{\mu}_n$ and standard deviation $\frac{\widehat{\sigma}_n}{\sqrt{n}}$.

ReallocateHistogram: The subroutines for the wholesale and piecemeal approaches are very similar. The only difference is that the reallocation of bucket frequences occurs within the subinterval $[\widehat{\mu}_n - \frac{\widehat{\sigma}_n}{\sqrt{n}}, \widehat{\mu}_n + \frac{\widehat{\sigma}_n}{\sqrt{n}}]$.

Quantiles: The details are the same as for extrema.

3.2 Experiments

We ran an extensive set of experiments to understand the following questions:

- How useful is it to use a summary data structure (in this case histograms) to maintain correlated aggregates as opposed to employing a "memoryless" algorithm? We ascertain the answer to this question by investigating the behavior a simple heuristic compared to histogram-based methods.
- How do our proposed methods compare with known histogram techniques, in particular, equiwidth and equidepth histograms? We demonstrate the impact of designing methods that specifically deal with the focused subranges involved in answering correlated aggregates, rather than for the *a priori* fixed ranges that existing histogram methods consider.
- Which approaches work well for our techniques? We have considered two general strategies, wholesale and piecemeal, and within each we have considered uniform and quantiled bucketing policies.

First we explain the experimental setup. Then we consider the answers to these questions in the context of the subsections that report on the accuracy for extrema and AVG as the independent aggregate.

3.2.1 *Experimental Setup*

Data: We used two real data sets: USAGE, usage data of 20K customers from AT&T; and MGCTY, lat/long of 65K road crossings in Montgomery County, MD.[3] We also used two synthetic data sets: ZIPF, a Zipfian distribution of points with $\lambda = 7$; and MULTIFRAC, a binomial multifractal obeying the "80-20 law".[4] The real data sets are ordered the way they were originally obtained; the synthetic data sets were generated in random order.

Queries: We tested correlated queries with both monotonic and non-monotonic independent aggregates. In particular, for monotonic aggregates we used MIN as the independent aggregate with the predicate $x < (1 + \epsilon) * \texttt{MIN}(x)$. For non-monotonic queries we used AVG as the independent aggregate with the predicate $x > \texttt{AVG}(x)$. We used COUNT and SUM as the dependent aggregates in both cases.

Quality Measure: To measure the accuracy, we used the formula given in Section 2:

$$RMSE_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(S_{out}[i] - S_{exact}[i])^2}$$

Competing Methods: As a frame of reference, we used two simple heuristics to maintain a running independent aggregate value and either (i) reset the count or (ii) continue to add to the existing one, when a new extrema value is encountered; this gives a lower- and upper-bound on the exact count, respectively. Among the existing methods, we computed "true" equiwidth and equidepth histograms, which required a single pass and multiple passes, respectively, at each time step. Clearly, this is not feasible in practice – we have given them an unfair advantage.[5] For our techniques, we consider both the wholesale and piecemeal strategies, each

[2] While our discussion uses a confidence interval of one standard deviation, this is a tunable parameter.

[3] Available at `http://www.esri.com/data/online/tiger`.

[4] Recent studies of network traffic data have shown that they are modeled well using multifractals [11].

[5] Note that the recent single-pass approximate quantile algorithms of [2, 20, 21] are designed for *offline* computation and, in any case, would likely give less accurate results than an exact equidepth histogram.

with uniform and quantiling partitioning policies. We call our methods `wholesale-uniform`, `wholesale-quantile`, `piecemeal-uniform`, and `piecemeal-quantile`.

3.2.2 Accuracy for Independent Extrema

We computed correlated aggregates of the form $\texttt{COUNT}\{y : x < (1+\epsilon) * \texttt{MIN}(x)\}$ and $\texttt{SUM}\{y : x < (1+\epsilon) * \texttt{MIN}(x)\}$ over real and synthetic data sets. Figure 4 plots the correlated `COUNT` at different time steps of a landmark window along with the streaming approximations determined by the competing methods, for the data sets USAGE and ZIPF. Similar plots were obtained for the other data sets but are omitted due to space constraints. The graphs in Figure 4(a) show the query answers for all of the methods. There is a clear separation of the methods. As expected, the simple heuristics give lower- and upper-bounds of the exact query; both were worse than all other methods. Also as expected, the equidepth histogram outperforms equiwidth. This was the case in all the experiments, so we drop equiwidth and report only the results from the equidepth method in the remaining plots. Note that the equidepth histogram appears to be diverging from the exact value over time, whereas the proposed methods track it closely. The graph in Figure 4(b) confirms this, plotting the error for the proposed methods. Figure 4(c) plots $RMSE_i$ for the proposed methods and for equidepth histograms using ZIPF. In both data sets, all of our methods give very small $RMSE_i$ (less than 5) and appear to stabilize at a constant approximation error. Figure 5 gives analogous graphs for the case where `SUM` is the dependent aggregate. Here we see an even greater divergence between equidepth histograms and the proposed methods.

3.2.3 Sensitivity Analysis for Extrema

To test the robustness of the methods, we performed the following experiments. First, we ran queries over the same data sets with different arrival orders. We tried several random permutations of the real data and found the results to be very similar to those in Figure 4; hence, we omit the plots for brevity. Then we artificially permuted the data so that initially only large values occur and there is a sudden large drop in the running minima. We present the plots from the USAGE data set with this partially-sorted reverse ordering in Figure 6. The error for the equidepth method appears to be increasing whereas it is decreasing for the other methods. Thus, our methods appear to be the most robust.

We varied the number of histogram buckets and ran the same experiments. Figure 7 plots the results with 5 buckets for USAGE; equidepth was chosen as a representative of the competing methods. Again, our methods gave the best accuracy. Figure 7(b) plots the RMSE for only our methods. Using 5 buckets rather than 10, we see a separation of the curves, with `piecemeal-uniform` performing the best, the wholesale approaches in the middle, and `piecemeal-quantile` performing the worst.

3.2.4 Accuracy for Independent Average

We computed correlated aggregates of the form $\texttt{COUNT}\{y : x > \texttt{AVG}(x)\}$ and $\texttt{SUM}\{y : x > \texttt{AVG}(x)\}$ over real and synthetic data sets. Figure 8 plots the exact tracking value at different time steps of a landmark window along with the streaming approximations determined by the competing methods, for USAGE and MULTIFRAC, which were chosen as representatives of real and synthetic data. The plots in Figure 8(a) tracks the correlated `COUNT` for all the methods. Unlike when `MIN` was the independent aggregate, here we see that the simple heuristic performs well. This is because the mean converges early on for these data sets, so the running average is a good estimate of the true average. Of the competitors, the equidepth histogram again performed the best (although not as well as in the `MIN` case), so we chose it as a representative in the $RMSE_i$ plots. Our methods again performed better than equidepth, especially for the MULTIFRAC data. As Figure 8(c) shows, the $RMSE_i$ for equidepth grows to 180, whereas it remains below 30 for our methods with sublinear growth. Figure 9 gives analogous graphs for the case where `SUM` is the dependent aggregate; there is an even greater divergence between equidepth histograms and the proposed methods.

3.2.5 Sensitivity Analysis for Average

We tested the robustness of the methods using the same experiments as we did for queries involving `MIN` as the independent aggregate. First, we tried several random permutations of the data and found that the accuracy for all the methods was slightly better (We omit the plot since the curves looked similar.) This is not surprising – the random arrival order of tuples helps the mean converge faster, and many of the methods are based on convergence properties of the mean. Then we artificially permuted the USAGE data so that initially only large values occur and there is a sudden large drop in values. Thus, the running mean will drop sharply at that point. Figure 10(a) presents the tracking value for all the methods; (b) presents the RMSE for equidepth and our methods. It is apparent that all methods gave worse accuracy overall. Our methods did not perform as well as the true equidepth method, as shown in Figure 10(b), due to the mean converge assumption built into them; however, they were clearly superior to equiwidth histograms. We also tried running the experiments with varying numbers of buckets, but the plots did not reveal any new observations.

4. SLIDING WINDOW ALGORITHMS

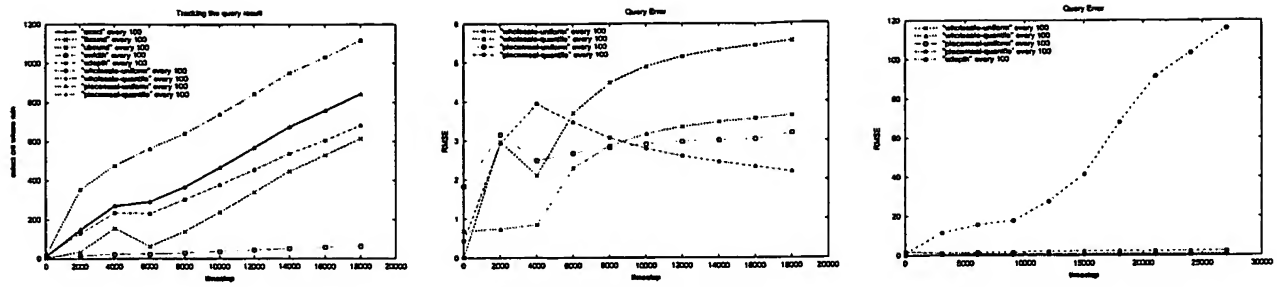
For sliding window aggregates, many of the properties discussed in Section 2 do not hold. In particular, extremas are not monotonic and confidence intervals about the mean do not converge like they do in the landmark window case. Thus, we use some reasonable heuristics in our algorithms. We describe these and present some experimental results to validate our techniques.

4.1 Algorithms

First, we give the outline of our algorithms, and then we describe the details of each method.

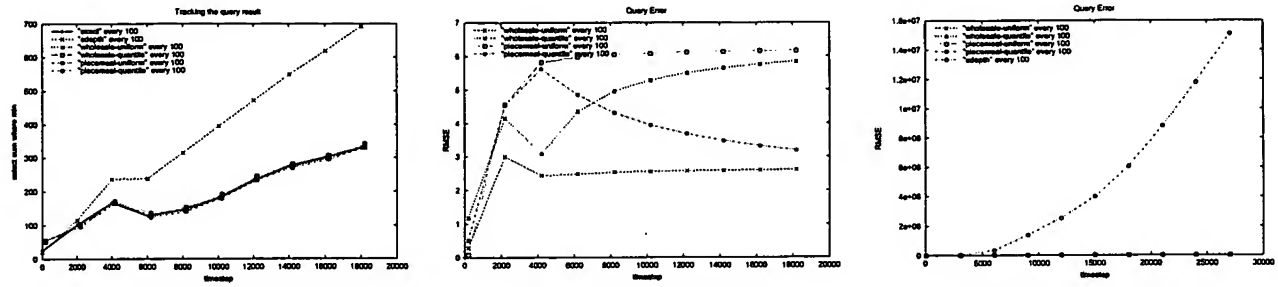
4.1.1 Sketch of Our Algorithms

Our algorithms all follow the general outline given in Figure 11. Again, the histograms contain m bins and are of the form $\langle (v_1, f_1), (v_2, f_2), ..., (v_m, f_m) \rangle$. The UpdateExtrema subroutine requires explanation. Unlike the landmark window case, where updating an extrema value is trivial, here the extrema values are non-monotonic because changes can occur due to both incoming and outgoing tuples. Thus, we must keep track of some auxiliary information to give good estimates for the extrema. We propose the following strategy for maintaining extrema values. We partition the sliding window into fixed-length intervals and keep track of the lo-



(a) tracking value for USAGE (b) $RMSE_i$ for USAGE (c) $RMSE_i$ for ZIPF

Figure 4: Correlated COUNT with independent MIN over landmark window: (a) tracking the query answer for USAGE with 10 buckets and $\epsilon = 99$; (b) $RMSE_i$; (c) $RMSE_i$ for ZIPF with 10 buckets and $\epsilon = 1000$.



(a) tracking value for USAGE (b) $RMSE_i$ for USAGE (c) $RMSE_i$ for ZIPF

Figure 5: Correlated SUM with independent MIN over landmark window: (a) tracking the query answer for USAGE with 10 buckets and $\epsilon = 99$; (b) $RMSE_i$; (c) $RMSE_i$ for ZIPF with 10 buckets and $\epsilon = 1000$.

cal extrema within each interval. When an outgoing (global) extrema value departs from the sliding window, we update the extrema using the remaining local extrema.

Sliding Window Algorithm:
Input: data stream and aggregate query
Output: result stream
$H \leftarrow$ InitializeHistogram(m);
while stream not empty do
read tuple $S_{in}[i]$;
UpdateExtrema($S_{in}[i]$);
if (*condition*) then ReallocateHistogram(H);
add incoming tuple to appropriate bucket;
delete outgoing tuple from appropriate bucket;

Figure 11: Sliding Window Algorithm

4.1.2 *Extrema as the Independent Aggregate*

Here we must keep track of an interval wider than the interval $[a, b] = [\min, (1+\epsilon) * \min]$ for the landmark window algorithm, since the extrema values are non-monotonic over a sliding window. We propose to place bins at $(\min, \ldots, (1+\epsilon) * maxmin, \max)$, where $maxmin$ is the maximum of the local minimums. The remaining bucket locations are placed according to the given partitioning policy.

InitializeHistogram: This subroutine is basically the same as the one for landmark windows.

PartitionHistogram: This subroutine partitions the histogram uniformly in the interval $[\min, (1+\epsilon) * maxmin]$, leaving one extra bucket from the end of this interval to MAX(x).

ReallocateHistogram: This subroutine is similar to the landmark window version, except that all but the last bucket are redistributed. In the case of the piecemeal approach, we approximately maintain a uniform partitioning of the buckets in $[\min, (1+\epsilon) * maxmin]$.

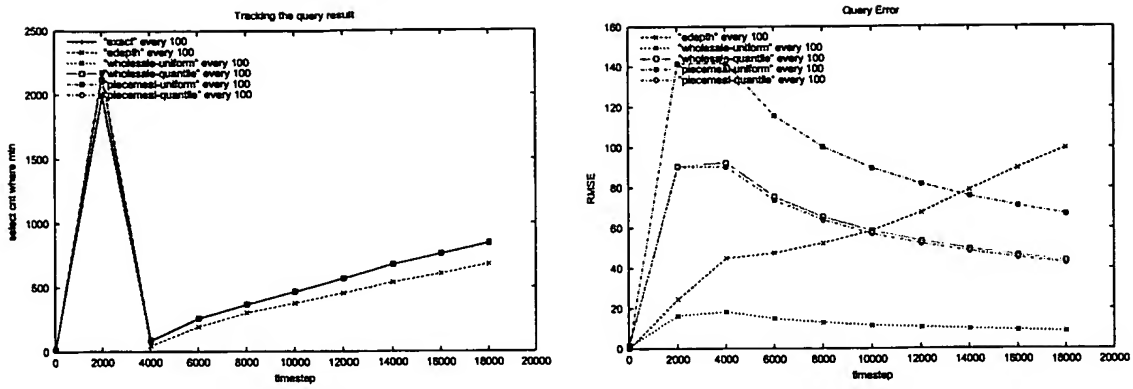
Quantiles: This subroutine is very similar to the landmark window version, except that all but the last bucket are requantiled.

4.1.3 *Average as the Independent Aggregate*

The algorithms are basically the same as the landmark window versions, except that the confidence interval does not shrink. Instead, it stays constant at $[\widehat{\mu}_n - \frac{\widehat{\sigma}_n}{\sqrt{w}}, \widehat{\mu}_n + \frac{\widehat{\sigma}_n}{\sqrt{w}}]$, where w is the size of the sliding window.

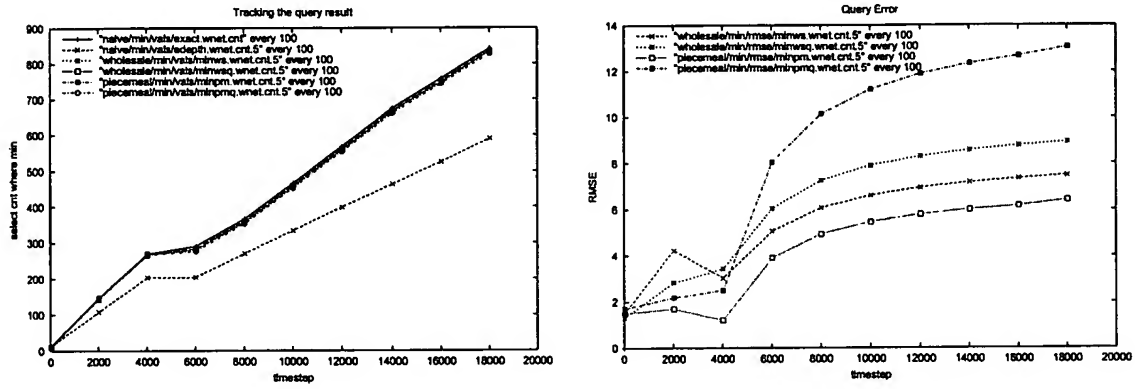
4.2 Experiments

The experimental setup is very similar to the one used in the landmark window case. We used the same data sets and queries. The competing methods are the sliding window versions of the landmark methods. We used the following



(a) query results (b) $RMSE_i$

Figure 6: Correlated COUNT with independent MIN over landmark window with data in partially-sorted reverse order: (a) tracking the query answer on USAGE with 10 buckets and $\epsilon = 99$; (b) $RMSE_i$.



(a) tracking value with 5 bins (b) $RMSE_i$ with 5 bins

Figure 7: Using fewer buckets: (a) query answer and (b) $RMSE_i$ for USAGE data with 5 buckets and $\epsilon = 99$.

quality measure for sliding window aggregates:

$$RMSE_n = \sqrt{\frac{1}{w}\sum_{i=n-w}^{n}(S_{out}[i] - S_{exact}[i])^2}$$

where w is the size of the window.

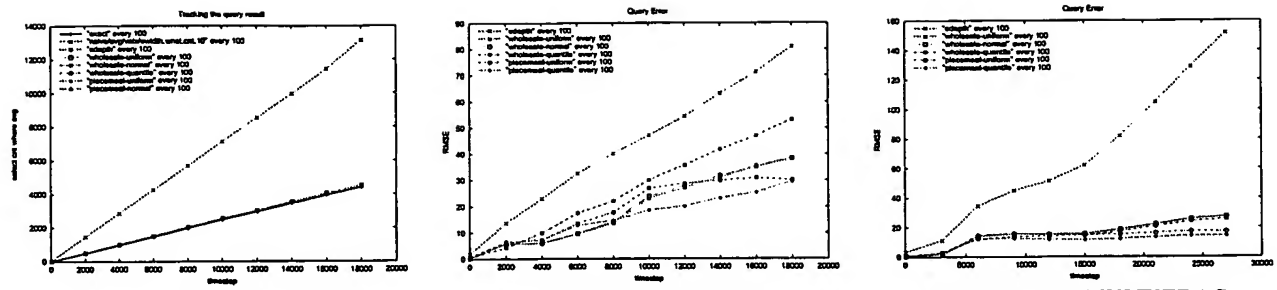
4.2.1 *Accuracy for Independent Extrema*

We computed correlated aggregates of the form $\texttt{COUNT}\{y : x < (1+\epsilon)*\texttt{MIN}(x)\}$ over real and synthetic data sets, over a sliding window of size 500. Figure 12 plots the exact tracking value at different time steps of a cumulative window along with the streaming approximations determined by the competing methods, for two representative data sets USAGE and MULTIFRAC. The plots in Figure 12(a) and (c) show the query result values for all the competing methods. It is somewhat clear that the best methods are equidepth and `piecemeal-uniform`. The plots in Figure 12(b) and (d) confirm this, showing the error for the proposed methods along with the equidepth competitor. In both plots, the RMSE accuracy of `piecemeal-uniform` is comparable to that of equidepth histograms. Note that, as we mentioned in Section 3, we have implemented a "true" (offline) equidepth histogram, requiring multiple passes over the data at each step; which is certainly no more feasible for sliding window streams than it is for landmark window streams.

The experiments we ran clearly separated out our methods. The versions of our methods with quantiled partitionings performed the worse. Since extrema values are not monotonic over sliding windows, the frequent and abrupt changes in bucket frequencies causes the quantiles to become stale quickly, thus rendering the policy useless. On the other hand, the methods which use uniform partitionings performed much better, with the piecemeal approach being superior to the wholesale approach in our experiments.

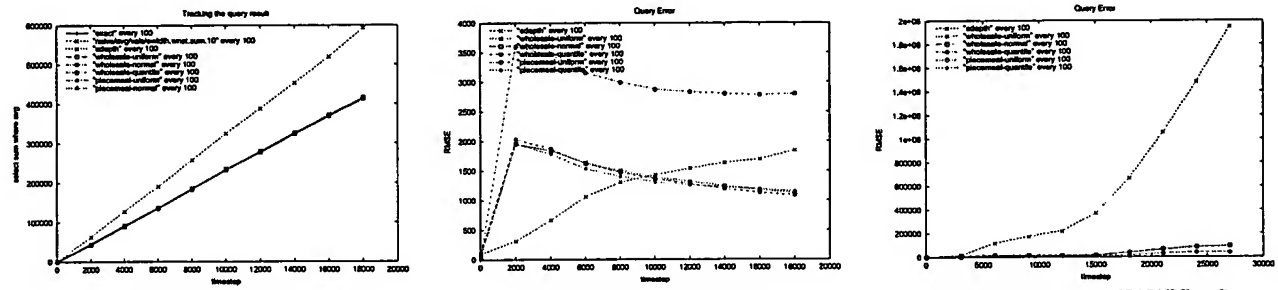
4.2.2 *Accuracy for Independent Average*

We computed correlated aggregates of the form $\texttt{COUNT}\{y : x > \texttt{AVG}(x)\}$ over real and synthetic data sets, over a sliding window of size 500. Figure 13 plots the exact tracking value at different time steps of a cumulative window along with the streaming approximations determined by the competing methods, for two representative data sets ZIPF and



(a) tracking value for USAGE (b) $RMSE_i$ for USAGE (c) $RMSE_i$ for MULTIFRAC

Figure 8: Correlated COUNT with independent AVG over landmark window: (a) tracking the query answer for the USAGE data with 10 buckets; (b) $RMSE_i$; (c) $RMSE_i$ for MULTIFRAC with 10 buckets.



(a) tracking value for USAGE (b) $RMSE_i$ for USAGE (c) $RMSE_i$ for MULTIFRAC

Figure 9: Correlated SUM with independent AVG over landmark window: (a) tracking the query answer for the USAGE data with 10 buckets; (b) $RMSE_i$; (c) $RMSE_i$ for MULTIFRAC with 10 buckets.

MGCTY. The plots in Figure 13(a) and (c) show the query result values for all the competing methods. The equidepth was slightly better in both experiments; the plots in Figure 13(b) and (d) show this. However, the proposed methods were competitive in both cases. For the ZIPF data, both wholesale methods were able to correct themselves after initially starting off with high $RMSE_i$.

The experiment on the real data set shows that the methods which employ uniform bucket partitioning are somewhat superior to the quantiled case, as we observed with sliding window queries with MIN as the independent aggregate. Here the mean is non-monotonic but, unlike with landmark windows, does not converge over time because it is computed over a fixed-length interval. Once again, the methods based on uniform partitioning appear to be more robust, making them more suitable to handle non-monotonicity.

5. RELATED WORK

Data streams have been of much recent interest [19]. In particular, algorithms and systems have been proposed for the computation of approximate frequency moments [1], L^1 and L^p distance functions [9, 12], and property testing [10].

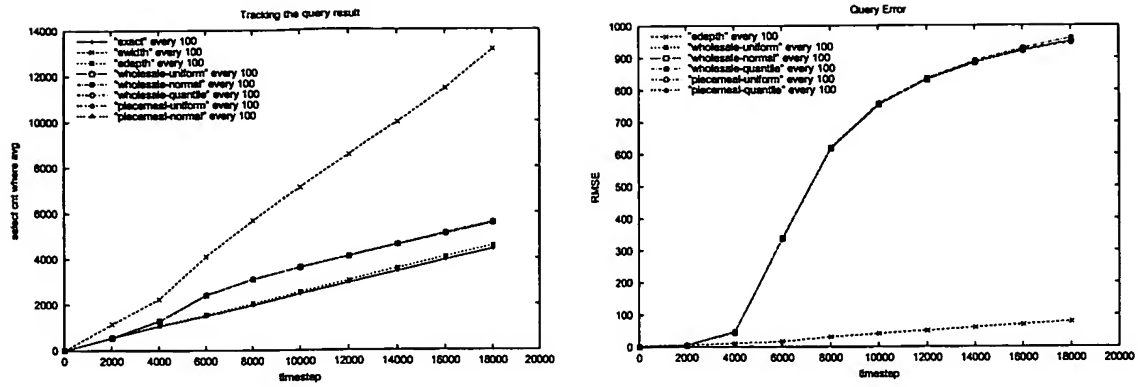
The maintenance of aggregate queries is a special case of the problem of incremental view maintenance; in particular, the maintenance of basic statistical aggregates in the presence of database updates was considered in [22]. The synopsis data structures of Matias et al. [15] consider the approximate maintenance of more fancy aggregates in the presence of updates. In online aggregation, Hellerstein et al. study the convergence of basic aggregates over finite data sets [18] and they describe access methods that retrieve records in random order in order to use statistical estimators based on independence assumptions. This work has been extended to online computation of joins [17], online reordering [23] and to adaptive query processing [3].

There has also been recent work on mining data streams, such as the construction of decision trees over data streams [13, 8] and clustering data streams [16]. Recent work by Alsabti et al. [2] and Manku et al. [20, 21] considers how to compute the approximate median and other quantiles in a single pass over a data set.

Correlated aggregates were originally considered in [6, 5, 4]; there the focus is on exact computation over finite data sets in multiple passes. To the best of our knowledge, there has not been any prior work on the approximate computation of this important class of aggregates.

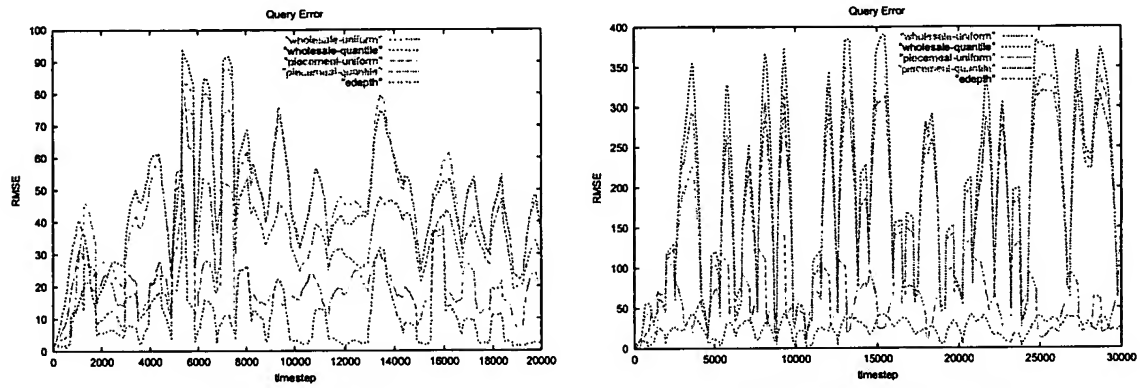
6. CONCLUSIONS AND FUTURE WORK

Effectively dealing with large volumes of streaming data, generated by applications as diverse as network management and telephone fraud detection, is a major challenge for the database community. A fundamental problem in this area is to compute and maintain a variety of complex aggregates, in an online fashion. We show that, while computing complex correlated aggregates exactly is not possible on streaming data, it is certainly feasible to do so *approximately*.



(a) query results (b) $RMSE_i$

Figure 10: Correlated COUNT with independent AVG over landmark window with data in partially-sorted reverse order: (a) tracking the query answer on USAGE with 10 buckets; (b) $RMSE_i$.



(a) $RMSE_i$ for USAGE (b) $RMSE_i$ for MULTIFRAC

Figure 12: Correlated COUNT with MIN as independent aggregate over sliding window of size $w = 500$: (a) $RMSE_i$ for USAGE with 10 buckets and $\epsilon = 99$; (b) $RMSE_i$ for MULTIFRAC with 10 buckets and $\epsilon = 99$.

Our solution to this problem is based on the use of focused histograms, which require accurate maintenance of summary information only in small data intervals; what makes the problem challenging is that these intervals are not known *a priori*: they can "move around", "expand" or "shrink", depending on the data in the stream. This renders current histogramming techniques ineffective. We presented two families of adaptive techniques, which we called wholesale and piecemeal, for efficiently "tracking" the true values of the desired correlated aggregates. Experimental results on a variety of real and synthetic data sets confirm the versatility of these techniques. Piecemeal, in particular, is a simple, elegant strategy that is extremely effective for a wide range of window scopes, and different types of aggregates, making it the strategy of choice.

7. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS: Journal of Computer and System Sciences*, 58, 1999.

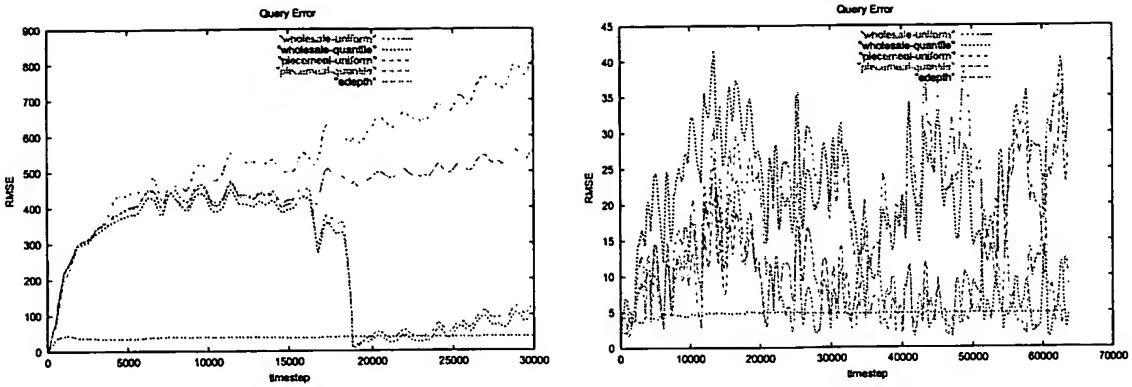
[2] K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 346–355, 1997.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 261–272, 2000.

[4] D. Chatziantoniou. Ad hoc OLAP: Expression and evaluation. In *Proceedings of the IEEE International Conference on Data Engineering*, 1999.

[5] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-join: An operator for complex OLAP. In *Proceedings of the IEEE International Conference on Data Engineering*, 2001.

[6] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the International Conference on Very*



(a) $RMSE_i$ for ZIPF (b) $RMSE_i$ for MGCTY

Figure 13: Correlated COUNT with AVG as independent aggregate over sliding window of size $w = 500$: (a) $RMSE_i$ for ZIPF with 10 buckets; (b) $RMSE_i$ for MGCTY with 10 buckets.

Large Databases, pages 295–306, 1996.

[7] A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors. *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*. ACM Press, 1999.

[8] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 71–80, Boston, MA, August 2000. ACM.

[9] J. Feigenbaum, R. Kannan, M. Strauss, and M. Viswanathan. An approximate L1-difference algorithm for massive data streams. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[10] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and spot-checking of data streams. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[11] A. Feldmann, A. Gilbert, and W. Willinger. Data networks as cascades: Investigating the multifractal nature of internet wan traffic. In *ACM SIGCOMM*, pages 42–55, 1998.

[12] J. Fong and M. Strauss. An approximate L^p-difference algorithm for massive data streams. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 2000.

[13] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. Boat-optimistic decision tree construction. In Delis et al. [7], pages 169–180.

[14] P. Gibbons, Y. Mattias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. *Proceedings of VLDB, Athens Greece*, pages 466–475, Aug. 1997.

[15] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 909–910, N.Y., Jan. 17–19 1999. ACM-SIAM.

[16] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *In Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE, November 2000.

[17] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 287–298, 1999.

[18] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182. ACM Press, 1997.

[19] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *Technical Report 1998-011, Digital Eqipment Corporation, Systems Research Center*, May, 1998.

[20] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 426–435. ACM Press, 1998.

[21] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In Delis et al. [7], pages 251–262.

[22] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 144–155. Morgan Kaufmann, 2000.

[23] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 709–720, 1999.

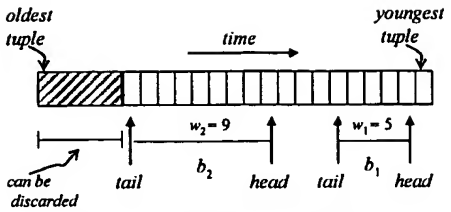


Figure 5: Queue organization

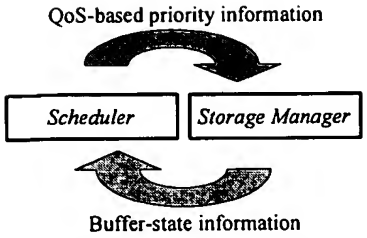


Figure 6: Scheduler-storage manager interaction

as new tuples are processed. ASM will keep track of these collections of pointers, and can normally discard tuples in a queue that are older than the oldest tail pointing into the queue. In summary, when a box produces a new tuple, it is added to the front of the queue. Eventually, all successor boxes process this tuple and it falls out of all of their windows and can be discarded. Figure 5 illustrates this model by depicting a two-way branch scenario where two boxes, b_1 and b_2, share the same queue ('w's refer to window sizes).

Normally, queues of this sort are stored as main memory data structures. However, ASM must be able to scale arbitrarily, and has chosen a different approach. Disk storage is divided into fixed length blocks, of a tunable size, *block_size*. We expect typical environment will use 128KB or larger blocks. Each queue is allocated one block, and queue management proceeds as above. As long as the queue does not overflow, the single block is used as a circular buffer. If an overflow occurs, ASM looks for a collection of two blocks (contiguous if possible), and expands the queue dynamically to $2 \times$ *block_size*. Circular management continues in this larger space. Of course, queue underflow can be treated in an analogous manner.

At start up time, ASM is allocated a buffer pool for queue storage. It pages queue blocks into and out of main memory using a novel replacement policy. The scheduler and ASM share a tabular data structure that contains a row for each box in the network containing the current scheduling priority of the box and the percentage of its queue that is currently in main memory. The scheduler periodically adjusts the priority of each box, while the ASM does likewise for the main memory residency of the queue. This latter piece of information is used by the scheduler for guiding scheduling decisions (see Section 4.3). The data structure also contains a flag to indicate that a box is currently running. Figure 6 illustrates this interaction.

When space is needed for a disk block, ASM evicts the lowest priority main memory resident block. In addition, whenever, ASM discovers a block for a queue that does not correspond to a running block, it will attempt to "upgrade" the block by evicting it in favor of a block for the queue corresponding to a higher priority box. In this way, ASM is continually trying to keep all the required blocks in main memory that correspond to the top priority queues. ASM is also aware of the size of each queue and whether it is contiguous on disk. Using this information, it can schedule multi-block reads and writes and garner added efficiency. Of course, as blocks move through the system and conditions change, the scheduler will adjust the priority of boxes, and ASM will react by adjusting the buffer pool. Naturally, we must be careful to avoid the well-known *hysteresis* effect, whereby ASM and the scheduler start working at cross purposes, and performance degrades sharply.

Connection Point Management. As noted earlier, the Aurora application designer indicates a collection of connection points, to which collections of boxes can be subsequently connected. This satisfies the Aurora requirement to support ad-hoc queries. Associated with each connection point is a history requirement and an optional storage key. The history requirement indicates the amount of historical information that must be retained. Sometimes, the amount of retained history is less than the maximum window size of the successor boxes. In this case, no extra storage need be allocated. The usual case is that additional history is requested.

In this case, ASM will organize the historical tuples in a B-tree organized on the storage key. If one is not specified, then a B-tree will be built on the timestamp field in the tuple. When tuples fall off the end of a queue that is associated with a connection point, then ASM will gather up batches of such tuples and insert them into the corresponding B-tree. Periodically, it will make a pass through the B-tree and delete all the tuples, which are older than the history requirement. Obviously, it is more efficient to process insertions and deletions in batches, than one by one.

Since we expect B-tree blocks to be smaller than *block_size*, we anticipate splitting one or more of the buffer pool blocks into smaller pieces, and paging historical blocks into this space. The scheduler will simply add the boxes corresponding to ad-hoc queries to the data structure mentioned above, and give these new boxes a priority. ASM will react by prefetching index blocks, but not data blocks, for worthy indexed structures. In turn, it will retain index blocks, as long as there are not higher priority buffer requirements. No attempt will be made to retain data blocks in main memory.

4.3 Real-Time Scheduling

Scheduling in Aurora is a complex problem due to the need to simultaneously address several issues including large system scale, real-time performance requirements, and dependencies between box executions. Furthermore, tuple processing in Aurora spans many scheduling and execution

steps (i.e., an input tuple typically needs to go through many boxes before potentially contributing to an output stream) and may involve multiple accesses to secondary storage. Basing scheduling decisions solely on QoS requirements, thereby failing to address end-to-end tuple processing costs, might lead to drastic performance degradation especially under resource constraints. To this end, Aurora not only aims to maximize overall QoS but also makes an explicit attempt to reduce overall tuple execution costs. We now describe how Aurora addresses these two issues.

Train Scheduling. In order to reduce overall processing costs, Aurora observes and exploits two basic *non-linearities* when processing tuples:

- *Inter-box non-linearity*: End-to-end tuple processing costs may drastically increase if buffer space is not sufficient and tuples need to be shuttled back and forth between memory and disk several times throughout their lifetime. One important goal of Aurora scheduling is, thus, to minimize tuple trashing. Another form of inter-box non-linearity occurs when passing tuples between box queues. If the scheduler can decide in advance that, say, box b_2 is going to be scheduled right after box b_1 (whose outputs feed b_2), then the storage manager can be bypassed (assuming there is sufficient buffer space) and its overhead avoided while transferring b_1's outputs to b_2's queue.
- *Intra-box non-linearity*: The cost of tuple processing may decrease as the number of tuples that are available for processing at a given box increases. This reduction in unit tuple processing costs may arise due to two reasons. First, the total number of box calls that need to be made to process a given number of tuples decreases, cutting down low-level overheads such as calls to the box code and context switch. Second, a box, depending on its semantics, may optimize its execution better with larger number of tuples available in its queue. For instance, a box can materialize intermediate results and reuse them in the case of windowed operations, or use merge-join instead of nested loops in the case of joins.

Aurora exploits the benefits of non-linearity in both inter-box and intra-box tuple processing primarily through *train scheduling*, a set of scheduling heuristics that attempt to (1) have boxes queue as many tuples as possible without processing—thereby generating long tuple trains; (2) process complete trains at once—thereby exploiting intra-box non-linearity; and (3) pass them to subsequent boxes without having to go to disk—thereby exploiting inter-box non-linearity. To summarize, train scheduling has two goals: its primary goal is to minimize the number of I/O operations performed per tuple. A secondary goal is to minimize the number of box calls made per tuple.

One important implication of train scheduling is that, unlike traditional blocking operators that wake up and process new input tuples as they arrive, Aurora scheduler tells each box when to execute and how many queued tuples to process. This somewhat complicates the implementation and increases the load of the scheduler, but is necessary for creating and processing tuple trains, which will significantly decrease overall execution costs.

Priority Assignment. The latency of each output tuple is the sum of the tuple's processing delay and its waiting delay. Unlike the processing delay, which is a function of input tuple rates and box costs, the waiting delay is primarily a function of scheduling. Aurora's goal is to assign priorities to outputs so as to achieve the per-output waiting delays that maximize the overall QoS.

The priority of an output is an indication of its urgency. Aurora currently considers two approaches for priority assignment. The first one, a *state-based* approach, assigns priorities to outputs based on their expected utility under the current system state, and then picks for execution, at each scheduling instance, the output with the highest utility. In this approach, the utility of an output can be determined by computing how much QoS will be *sacrificed* if the execution of the output is deferred. A second, *feedback-based* approach continuously observes the performance of the system and dynamically reassigns priorities to outputs, properly increasing the priorities of those that are not doing well and decreasing priorities of the applications that are already in their *good zones*.

Putting It All Together. Because of the large scale, highly dynamic nature of the system, and the granularity of scheduling, searching for optimal scheduling solutions is clearly infeasible. Aurora therefore uses heuristics to simultaneously address real-time requirements and cost reduction by first assigning priorities to select individual outputs and then exploring opportunities for constructing and processing tuple trains.

We now describe one such heuristic used by Aurora. Once an output is selected for execution, Aurora will find the first downstream box whose queue is in memory (note that for a box to be schedulable, its queue must at least contain its window's worth of tuples). Going upstream, Aurora will then consider other boxes, until either it considers a box whose queue is not in memory or it runs out of boxes. At this point, there is a sequence of boxes (i.e., a *superbox*) that can be scheduled one after another.

In order to execute a box, Aurora contacts the storage manager and asks that the queue of the box be pinned to the buffer throughout box's execution. It then passes the location of the input queue to the appropriate box processor code, specifies how many tuples the box should process, and assigns it to an available worker thread.

4.4 Introspection

Aurora employs static and run-time introspection techniques to predict and detect overload situations.

Static Analysis. The goal of static analysis is to determine if the hardware running the Aurora network is sized correctly. If insufficient computational resources are present to handle the steady state requirements of an Aurora network, then queue lengths will increase without bound and response times will become arbitrarily large.

As described before, each box b in an Aurora network has an expected tuple processing cost, $c(b)$, and a

selectivity, $s(b)$. If we also know the expected rate of tuple production $r(d)$ from each data source d, then we can use the following static analysis to ascertain if Aurora is sized correctly.

From each data source, we begin by examining the immediate downstream boxes: if box b_i is directly downstream from data source d_i, then, for the system to be stable, the throughput of b_i should be at least as large as the input data rate; i.e.,

$$1/c(b_i) \geq r(d_i)$$

We can then calculate the output data rate from b_i as:

$$min(1/c(b_i), r(d_i)) \times s(b_i)$$

Proceeding iteratively, we can compute the output data rate and computational requirements for each box in an Aurora network. We can then calculate the minimum aggregate computational resources required per unit time, *min_cap*, for stable steady-state operation. Clearly, the Aurora system with a capacity C cannot handle the expected steady state load if C is smaller than *min_cap*. Furthermore, the response times will assuredly suffer under the expected load of ad-hoc queries if

$$C \times H < min\_cap$$

Clearly, this is an undesirable situation and can be corrected by redesigning applications to change their resource requirements, by supplying more resources to increase system capacity, or by load shedding.

Dynamic Analysis. Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable, long-duration spikes in input rates may deteriorate performance to a level that renders the system useless. We now describe two run-time techniques to detect such cases.

Our technique for detecting an overload relies on the use of delay-based QoS information. Aurora timestamps all tuples from data sources as they arrive. Furthermore, all Aurora operators preserve the tuple timestamps as they produce output tuples (if an operator has multiple input tuples, then the earlier timestamp is preserved). When Aurora delivers an output tuple to an application, it checks the corresponding delay-based QoS graph (Figure 4a) for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone).

4.5 Load Shedding

When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora tuple processing via *load shedding*. The naïve approach to load shedding involves dropping tuples at random points in the network in an entirely uncontrolled manner. This is similar to dropping overflow packets in packet-switching networks [27], and has two potential problems: (1) overall system utility might be degraded more than necessary; and (2) application semantics might be arbitrarily affected. In order to alleviate these problems, Aurora relies on QoS information to guide the load shedding process. We now describe two load-shedding techniques that differ in the way they exploit QoS.

Load Shedding by Dropping Tuples. The first approach addresses the former problem mentioned above: it attempts to minimize the degradation (or maximize the improvement) in the overall system QoS; i.e., the QoS values aggregated over all the outputs. This is accomplished by dropping tuples on network branches that terminate in *more tolerant* outputs.

If load shedding is triggered as a result of static analysis, then we cannot expect to use delay-based or value-based QoS information (without assuming the availability of a priori knowledge of the tuple delays or frequency distribution of values). On the other hand, if load shedding is triggered as a result of dynamic analysis, we can also use delay-based QoS graphs.

We use a greedy algorithm to perform load shedding. Let us initially describe the static load shedding algorithm driven by drop-based QoS graphs. We first identify the output with the *smallest* negative slope for the corresponding QoS graph. We move horizontally along this curve until there is another output whose QoS curve has a smaller negative slope at that point. This horizontal difference gives us an indication of the *output* tuples to drop (i.e., the selectivity of the drop box to be inserted) that would result in the minimum decrease in the overall QoS. We then move the corresponding drop box as far upstream as possible until we find a box that affects other outputs (i.e., a *split point*), and place the drop box at this point. Meanwhile, we can calculate the amount of recovered resources. If the system resources are still not sufficient, then we repeat the process.

For the run-time case, the algorithm is similar except that we can use delay-based QoS graphs to identify the problematic outputs, i.e., the ones that are beyond their delay thresholds, and we repeat the load shedding process until the latency goals are met.

In general, there are two subtleties in dynamic load shedding. First, drop boxes inserted by the load shedder should be among the ones that are given higher priority by the scheduler. Otherwise, load shedding will be ineffective in reducing the load of the system. Therefore, the load shedder simply does not consider the *inactive* (i.e., low priority) outputs, which are indicated by the scheduler. Secondly, the algorithm tries to move the drop boxes as close to the sources as possible to discard tuples before they redundantly consume any resources. On the other hand, if there is a box with a large existing queue, it makes sense to *temporarily* insert the drop box at that point rather than trying to move it upstream closer towards the data sources.

Presumably, the application is coded so that it can tolerate missing tuples from a data source caused by communication failures or other problems. Hence, load shedding simply artificially introduces additional missing tuples. Although the semantics of the application are somewhat different, the harm should not be too damaging.

Semantic Load Shedding by Filtering Tuples. The load shedding scheme described above effectively reduces the amount of Aurora processing by dropping *randomly selected* tuples at strategic points in the network. While this

approach attempts to minimize the loss in overall system utility, it fails to control the impact of the dropped tuples on application semantics. Semantic load shedding addresses this limitation by using value-based QoS information, if available. Specifically, semantic load shedding drops tuples in a more controlled way; i.e., it drops less important tuples, rather than random ones, using filters.

If value-based QoS information is available, then Aurora can watch each output and build up a histogram containing the frequency with which value ranges have been observed. In addition, Aurora can calculate the expected utility of a range of outputs by multiplying the QoS values with the corresponding frequency values for every interval and then summing these values. To shed load, Aurora identifies the output with the *lowest utility interval*; converts this interval to a filter predicate; and then, as before, attempts to propagate the corresponding filter box as far upstream as possible to a split point. This strategy, which we refer to as *backward interval propagation*, admittedly has limited scope because it requires the application of the inverse function for each operator passed upstream (Aurora boxes do not necessarily have inverses). In an alternative strategy, *forward interval propagation*, Aurora starts from an output and goes upstream until it encounters a split point (or reaches the source). It then *estimates* a proper filter predicate and propagates it in downstream direction to see what results at the output. By trial-and-error, Aurora can converge on a desired filter predicate. Note that a combination of these two strategies can also be utilized. First, Aurora can apply backward propagation until a box, say *b*, whose operator's inverse is difficult to compute. Aurora can then apply forward propagation between the insertion location of the filter box and *b*. This algorithm can be applied iteratively until sufficient load is shed.

5 Related Work

A special case of Aurora processing is as a continuous query system. A system like Niagara [7] is concerned with combining multiple data sources in a wide area setting, while we are initially focusing on the construction of a general stream processor that can process very large numbers of streams.

Query indexing [3] is an important technique for enhancing the performance of large-scale filtering applications. In Aurora, this would correspond to a merge of some inputs followed by a fanout to a large number of filter boxes. Query indexing would be useful here, but it represents only one Aurora processing idiom.

As in Aurora, active databases [21, 22] are concerned with monitoring conditions. These conditions can be a result of any arbitrary update on the stored database state. In our setting, updates are append-only, thus requiring different processing strategies for detecting monitored conditions. Triggers evaluate conditions that are either true or false. Our framework is general enough to support queries over streams or the conversion of these queries into monitored conditions. There has also been extensive work on making active databases highly scalable (e.g., [11]). Similar to continuous query research, these efforts have focused on query indexing, while Aurora is constructing a more general system.

Adaptive query processing techniques (e.g., [4, 13, 26]) address efficient query execution in unpredictable and dynamic environments by revising the query execution plan as the characteristics of incoming data changes. Of particular relevance is the Eddies work [4]. Unlike traditional query processing where every tuple from a given data source gets processed in the same way, each tuple processed by an Eddy is dynamically routed to operator threads for partial processing, with the responsibility falling upon the tuple to carry with it its processing state. Recent work [17] extended Eddies to support the processing of queries over streams, mainly by permitting Eddies systems to process multiple queries simultaneously and for unbounded lengths of time. The Aurora architecture bears some similarity to that of Eddies in its division of a single query's processing into multiple threads of control (one per query operator). However, queries processed by Eddies are expected to be processed in their entirety; there is neither the notion of load shedding, nor QoS.

Previous work on stream data query processing architectures shares many of the goals and target application domains with Aurora. The Streams project [5] attempts to provide complete DBMS functionality along with support for continuous queries over streaming data. The Fjords architecture [16] combines querying of push-based sensor sources with pull-based traditional sources by embedding the pull/push semantics into queues between query operators. It is fundamentally different from Aurora in that operator scheduling is governed by a combination of schedulers specific to query threads and operator-queue interactions. Tribeca [25] is an extensible, stream-oriented data processor designed specifically for supporting network traffic analysis. While Tribeca incorporates some of the stream operators and compile-time optimizations Aurora supports, it does not address scheduling or load shedding issues, and does not have the concept of ad-hoc queries.

Work in sequence databases [24] defined sequence definition and manipulation languages over discrete data sequences. The Chronicle data model [14] defined a restricted view definition and manipulation language over append-only sequences. Aurora's algebra extends the capabilities of previous proposals by supporting a wider range of window processing (i.e., Tumble, Slide, Latch), classification (i.e., GroupBy), and interpolation (i.e., Resample) techniques.

Our work is also relevant to materialized views [10], which are essentially stored continuous queries that are re-executed (or incrementally updated) as their base data are modified. However, Aurora's notion of continuous queries differs from materialized views primarily in that Aurora updates are append-only, thus, making it much easier to incrementally materialize the view. Also, query results are streamed (rather than stored); and high stream data rates may require load shedding or other approximate query

processing techniques that trade off efficiency for result accuracy.

Our work is likely to benefit from and contribute to the considerable research on temporal databases [20], main-memory databases [8], and real-time databases [15, 20]. These studies commonly assume an HADP model, whereas Aurora proposes a DAHP model that builds streams as fundamental Aurora objects. In a real-time database system, transactions are assigned timing constraints and the system attempts to ensure a degree of confidence in meeting these timing requirements. The Aurora notion of QoS extends the soft and hard deadlines used in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual transactions, whereas Aurora associates QoS curves with outputs from stream processing and, thus, has to support continuous timing requirements. Relevant research in workflow systems (e.g., [18]) primarily focused on organizing long-running interdependent activities but did not consider real-time processing issues.

There has been extensive research on scheduling tasks in real-time and multimedia systems and databases [19, 20]. The proposed approaches are commonly deadline driven; i.e., at each scheduling point, the task that has the earliest deadline or one that is expected to provide the highest QoS (e.g., throughput) is identified and scheduled. In Aurora, such an approach is not only impractical because of the sheer number of potentially schedulable tasks (i.e., tuples), but is also inefficient because of the implicit assumption that all tasks are memory-resident and are scheduled and executed in their entirety. To the best of our knowledge, however, our train scheduling approach is unique in its ability to reduce overall execution costs by exploiting intra- and inter-box non-linearities described here.

The work of [26] takes a scheduling-based approach to query processing; however, they do not address continuous queries, are primarily concerned with data rates that are too slow (we also consider rates that are too high), and they only address query plans that are trees with single outputs.

The congestion control problem in data networks [27] is relevant to Aurora and its load shedding mechanism. Load shedding in networks typically involves dropping individual packets randomly, based on timestamps, or using (application-specified) priority bits. Despite conceptual similarities, there are also some fundamental differences between network load shedding and Aurora load shedding. First, unlike network load shedding which is inherently distributed, Aurora is aware of the entire system state and can potentially make more intelligent shedding decisions. Second, Aurora uses QoS information provided by the external applications to trigger and guide load shedding. Third, Aurora's semantic load shedding approach not only attempts to minimize the degradation in overall system utility, but also quantifies the imprecision due to dropped tuples.

Aurora load shedding is also related to approximate query answering (e.g., [12]), data reduction, and summary techniques [6, 9], where result accuracy is traded for efficiency. By throwing away data, Aurora bases its computations on sampled data, effectively producing approximate answers using data sampling. The unique aspect of our approach is that our sampling is driven by QoS specifications.

6 Implementation Status

As of June 2002, we have a prototype Aurora implementation. The prototype has a Java-based GUI that allows construction and execution of Aurora networks. The interface is currently primitive, but will be extended over the next few months to support specification of QoS graphs, connection points, and zoom. The run-time system contains a primitive scheduler, a rudimentary storage manager, and code to execute most of the boxes. Aurora metadata is stored in a schema, which is stored in a Berkeley DB [1] database. Hence, Aurora is functionally complete, and multi-box networks can be constructed and run. However, there is currently no optimizer and load shedding. We expect to implement Aurora functionality in these areas over the course of the summer.

7 Conclusions and Future Work

Monitoring applications are those where streams of information, triggers, real-time requirements, and imprecise data are prevalent. Traditional DBMSs are based on the HADP model, and thus cannot provide adequate support for such applications. In this paper, we have described the architecture of Aurora, a DAHP system oriented towards monitoring applications. We argued that providing efficient support for these demanding applications not only require critically revisiting many existing aspects of database design and implementation, but also require developing novel proactive data storage and processing concepts and techniques.

In this paper, we first presented the basic Aurora architecture, along with the primitive building blocks for workflow processing. We followed with several heuristics for optimizing a large Aurora network. We then focused on run-time data storage and processing issues, discussing storage organization, real-time scheduling, introspection, and load shedding, and proposed novel solutions in all these areas.

We are currently implementing an Aurora prototype system, which we will use to investigate the practicality and efficiency of our proposed solutions. We are also investigating two important research directions. While the bulk of the discussion in this paper describes how Aurora works on a single computer, many stream-based applications demand support for distributed processing. To this end, we are working on a distributed architecture, Aurora*, which will enable operators to be pushed closer to the data sources, potentially yielding significantly improved scalability, energy use, and bandwidth efficiency. Aurora* will provide support for distribution by running a full Aurora system on each of a collection of communicating nodes. In particular, Aurora* will manage load by replicating boxes along a path and migrating a copy